

ПРОГРАММИРОВАНИЕ В C++Builder 5

А.Я. Архангельский

Простое и быстрое построение
приложений для Windows

Интернационализация

Серверы COM

Технологии доступа к данным
ADO и InterBase Express

Работа с данными любых
типов

Детальное описание
компонентов

Методические и справочные
материалы по C++Builder 5,
C++, API Windows

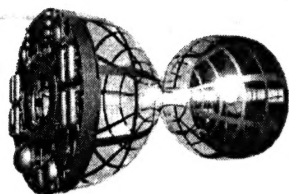


БИНОМ









ПРОГРАММИРОВАНИЕ В **C++ Builder 5**

А.Я. Архангельский

ПРОГРАММИРОВАНИЕ В C++ Builder 5



Москва
ЗАО «Издательство БИНОМ»
2002

УДК 004.43
ББК 32.973.26-018.1
А87

А.Я. Архангельский

Программирование в C++Builder 5. — М.: ЗАО «Издательство БИНОМ», 2002 г. — 1152 с.: ил.

Книга содержит методические и справочные материалы по новой версии широко известной системы визуального объектно-ориентированного программирования C++Builder. Рассмотрена методика построения прикладных программ, реализующих текстовые и графические редакторы, мультипликацию и мультимедиа, работу с базами данных, построение справочных систем, отчетов, интерфейсов к внешним программам. Обширная справочная часть книги содержит материалы по языку C++, функциям C++Builder и API Windows (свыше 570 функций), типам и классам C++Builder (около 200), их свойствам, методам и событиям (около 400).

В книге рассмотрено много возможностей и инструментов, впервые введенных в новой версии C++Builder 5.

Книга рассчитана как на начинающих, владеющих только основами какого-нибудь языка программирования, так и на опытных разработчиков.

ISBN 5-7989-0191-2

© Архангельский А.Я., 2002

© ЗАО «Издательство БИНОМ», 2002

Научно-популярное издание

А.Я. Архангельский

Программирование в C++Builder 5

Компьютерная верстка *С.В. Лычагина, К.А. Свиридова*

Подписано в печать 14.02.2002. Формат 70×100/16. Усл. печ. л. 93,6.

Гарнитура Школьная. Бумага газетная. Печать офсетная.

Тираж 3000 экз. Заказ 611

ЗАО «Издательство БИНОМ», 2002 г.

103473, Москва, Краснопролетарская, 16

Лицензия на издательскую деятельность № 065249 от 26 июня 1997 г.

Отпечатано с готовых диапозитивов
в ФГУП ордена Трудового Красного Знамени «Техническая книга»
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций
198005, Санкт-Петербург, Измайловский пр., 29.

Содержание

От автора	17
Для кого и о чем эта книга	17
Отличие от книг серии «Все о C++Builder» и «Программирование в C++Builder 4».	18
Что вы найдете в этой книге.	19
Чего вы не найдете в этой книге	21
Рекомендации по работе с книгой	21
Благодарности	22
 ЧАСТЬ 1. Объектно-ориентированное визуальное программирование	 23
 Глава 1. Принципы объектно-ориентированного визуального программирования	 25
1.1 Объектно-ориентированное программирование	25
1.2 Основы визуального программирования интерфейса	28
1.3 Предварительные сведения о классах и наследовании	29
1.4 Системы быстрой разработки приложений	30
1.5 Язык объектно-ориентированного проектирования C++	31
1.5.1 Введение	31
1.5.2 Общие сведения о программах на C++	31
1.5.3 Структура головного файла проекта.	32
1.5.4 Структура файлов модулей форм	36
1.5.5 Области видимости и доступ к объектам, переменным и функциям модуля	38
1.5.6 Работа с указателями на объекты.	47
 Глава 2. Система визуального объектно-ориентированного программирования C++Builder	 53
2.1 Что может C++Builder 5.	53
2.2 Интегрированная Среда Разработки (ИСР) C++Builder.	54
2.2.1 Общий вид окна ИСР	54
2.2.2 Полоса главного меню и всплывающие меню	55
2.2.3 Быстрые кнопки.	56
2.2.4 Палитра компонентов	57
2.2.5 Окно формы.	59
2.2.6 Окно Редактора Кода	59
2.2.7 Инспектор Объектов	61
2.2.8 Перетаскивание и встраивание окон в ИСР C++Builder.	64
2.2.9 Управление конфигурациями окон ИСР.	65
2.3 Первые шаги — первые собственные приложения	67
2.3.1 Очень простое приложение	67
2.3.2 Немного более сложное приложение	69
2.4 Проекты C++Builder	71
2.4.1 Организация проекта в C++Builder, основные файлы проектов	71
2.4.2 Создание и сохранение нового проекта	74
2.4.3 Менеджер проектов	79
2.4.4 Управление проектами.	81
2.5 Основные проектные операции при создании приложения	85
2.5.1 Включение в проект новой формы	85
2.5.2 Размещение компонентов на форме	88

2.5.3 Инструментальные средства поддержки разработки кода	95
2.6 Отладка приложений	101
2.6.1 Компиляция и компоновка проекта	101
2.6.2 Сообщения компилятора и компоновщика	103
2.6.3 Что делать, если произошла ошибка выполнения	106
2.6.4 Окно наблюдения Watch List	108
2.6.5 Окно оценки и модификации Evaluate/Modify	110
2.6.6 Выполнение приложения по шагам	111
2.6.7 Точки прерывания	113
2.6.8 Использование окна Инспектора Отладки Debug Inspector	118
2.6.9 Протокол событий, функция OutputDebugString	120
2.6.10 Другие средства отладки	121
2.6.11 Некоторые приемы программирования, встраивающие отладку в код	121
Глава 3. Обзор компонентов библиотеки C++Builder	127
3.1 Страницы палитры компонентов	127
3.2 Компоненты ввода и отображения текстовой информации	129
3.2.1 Перечень компонентов ввода и отображения текстовой информации	129
3.2.2 Отображение текста в надписях компонентов Label, StaticText, Panel	131
3.2.3 Окна редактирования Edit и MaskEdit	133
3.2.4 Многострочные окна редактирования Memo и RichEdit	137
3.2.5 Компоненты выбора из списков — ListBox, CheckListBox, ComboBox	141
3.2.6 Таблица строк — компонент StringGrid	144
3.3 Ввод и отображение чисел, дат и времени	145
3.3.1 Перечень компонентов ввода и отображения чисел, дат и времени	145
3.3.2 Ввод и отображение целых чисел — компоненты UpDown и SpinEdit	146
3.3.3 Ввод и отображение дат и времени — компоненты DateTimePicker, MonthCalendar, Calendar	147
3.3.4 Страницы Excel — компонент F1Book	149
3.4 Компоненты отображения и ввода иных видов информации	150
3.4.1 Компоненты отображения иерархических данных — TreeView и Outline	150
3.4.2 Отображение информации в стиле папок Windows — компонент ListView	156
3.4.3 Отображение хода длительных процессов — компоненты ProgressBar и CGauge	160
3.4.4 Таблицы изображений — компоненты DrawGrid и StringGrid	162
3.4.5 Отображение форм — компонент Shape	163
3.4.6 Графики и диаграммы — компонент Chart	163
3.4.7 Графики и диаграммы — компоненты Chartfx и Graph	168
3.5 Кнопки, индикаторы, управляющие элементы	171
3.5.1 Управляющие кнопки Button и BitBtn	171
3.5.2 Кнопка с фиксацией SpeedButton	173
3.5.3 Группы радиокнопок — компоненты RadioGroup, RadioButton и GroupBox	174
3.5.4 Индикаторы CheckBox и CheckListBox	176
3.5.5 Ползунки и полосы прокрутки — компоненты TrackBar и ScrollBar	177
3.5.6 Таймер — компонент Timer	179
3.6 Компоненты — меню	180
3.6.1 Главное меню — компонент MainMenu	180
3.6.2 Контекстное всплывающее меню — компонент PopupMenu	184
3.6.3 Горячие клавиши — компонент HotKey	185
3.7 Панели и компоненты внешнего оформления	188
3.7.1 Общая характеристика	188
3.7.2 Панели общего назначения — компоненты Panel, GroupBox, Bevel, ScrollBox, Splitter	189
3.7.3 Заголовки — компоненты HeaderControl и Header	191
3.7.4 Многостраничные панели — компоненты TabControl, PageControl, TabSet, TabbedNotebook, Notebook	192

3.7.5	Инструментальные панели — компоненты ToolBar и PageScroller	195
3.7.6	Перестраиваемые панели — компоненты CoolBar и ControlBar	198
3.7.7	Полоса состояния StatusBar	200
3.7.8	Фреймы	201
3.8	Системные диалоги	208
3.8.1	Общая характеристика компонентов — диалогов	
3.8.2	Диалоги открытия и сохранения файлов — компоненты OpenDialog, SaveDialog, OpenPictureDialog, SavePictureDialog	209
3.8.3	Фрагменты диалогов — компоненты DriveComboBox, DirectoryListBox, FilterComboBox, FileListBox и CDirectoryOutline	214
3.8.4	Диалог выбора шрифта — компонент FontDialog	216
3.8.5	Диалог выбора цвета — компонент ColorDialog	218
3.8.6	Диалоги печати и установки принтера — компоненты PrintDialog и PrinterSetupDialog	219
3.8.7	Диалоги поиска и замены текста — компоненты FindDialog и ReplaceDialog	221
3.9	Компоненты организации управления приложением	226
3.9.1	Диспетчеризация событий — компоненты, связанные с ActionList	226
3.9.2	Список изображений — компонент ImageList	228
3.9.3	Приложение — компонент ApplicationEvents и объект Application	230
ЧАСТЬ 2. Разработка приложений для Windows		237
Глава 4. Проектирование графического интерфейса пользователя.		239
4.1	Требования к интерфейсу пользователя приложений для Windows	239
4.1.1	Общие рекомендации по разработке графического интерфейса	239
4.1.2	Многооконные приложения	240
4.1.3	Стиль окон приложения	240
4.1.4	Цветовое решение приложения	245
4.1.5	Шрифты текстов	247
4.1.6	Меню	248
4.1.7	Компоновка форм	253
4.1.8	Последовательность фокусировки элементов	254
4.1.9	Подсказки и контекстно-зависимые справки	256
4.2	Проектирование окон с изменяемыми размерами.	259
4.2.1	Выравнивание компонентов — свойство Align	259
4.2.2	Изменение местоположения и размеров компонентов	261
4.2.3	Панели с перестраиваемыми границами	263
4.2.4	Ограничение пределов изменения размеров окон и компонентов	264
4.2.5	Масштабирование компонентов	265
4.3	Обработка событий клавиатуры и мыши.	266
4.3.1	События мыши	266
4.3.2	События клавиатуры	271
4.4	Перетаскивание объектов.	275
4.4.1	Перетаскивание информации об объектах — технология Drag&Drop	275
4.4.2	Перетаскивание и встраивание объектов — Drag&Doc. Плавающие окна	279
4.4.3	Буксировка компонентов в окне приложения.	286
4.5	Формы	290
4.5.1	Управление формами	290
4.5.2	Модальные формы	295
4.5.3	Пример приложения с модальными формами заставки и запроса пароля	296
4.5.4	Управление формами в приложениях с интерфейсом множества документов (приложениях MDI)	299
4.5.5	Пример приложения с интерфейсом множества документов — простой многооконный редактор.	301
4.5.6	Объект Screen и приложения, работающие с несколькими мониторами	304
4.6	Печать в C++Builder	308
4.6.1	Печать форм методом Print	308

4.6.2 Методы компонентов, обеспечивающие печать	308
4.6.3 Печать средствами офисных приложений Windows с помощью функции ShellExecute и обращения к серверам COM	309
4.6.4 Печать с помощью объекта Printer.	309
4.7 Развертывание приложений	311
4.7.1 Оформление заверченного проекта	311
4.7.2 Установка и настройка приложения	321
Глава 5. Графика и мультимедиа	329
5.1 Построение графических изображений	329
5.1.1 Использование готовых графических файлов	329
5.1.2 Редактор Изображений Image Editor	334
5.1.3 Канва — холст для рисования	340
5.1.4 Пример построения собственного простого графического редактора	346
5.1.5 Режимы рисования	351
5.1.6 Продолжение создания собственного графического редактора	353
5.1.7 События OnPaint	360
5.2 Мультимедиа и анимация	362
5.2.1 Звук	362
5.2.2 Начала анимации — создание собственной мультипликации.	366
5.2.3 Воспроизведение немых видео клипов — компонент Animate	373
5.2.4 Универсальный проигрыватель MediaPlayer	376
Глава 6. Взаимодействие приложения с внешними программами	381
6.1 Запуск из приложения внешних программ.	381
6.1.1 Запуск внешней программы функцией execvp.	381
6.1.2 Запуск внешней программы функцией spawnlp	383
6.1.3 Запуск внешней программы функцией WinExec	385
6.1.4 Запуск внешней программы и открытие документа функцией ShellExecute	387
6.2 Управление внешними приложениями	390
6.2.1 Определение дескриптора окна приложения	390
6.2.2 Некоторые функции API Windows для управления окнами	392
6.3 Сообщения Windows и их обработка.	392
6.3.1 Обработка сообщений в приложениях C++Builder.	392
6.3.2 Посылка сообщений	394
6.3.3 Обработка сообщений	396
6.3.4 Определение собственных сообщений	400
6.4 Внедрение и связывание объектов — OLE	401
6.4.1 Общие сведения	401
6.4.2 Внедрение и связывание	403
6.4.3 Автоматизация OLE	407
6.4.4 Компоненты — серверы COM в C++Builder 5	410
6.5 Динамический обмен данными — DDE	422
6.5.1 Общие сведения	422
6.5.2 Установление контакта с сервером.	423
6.5.3 Обмен данными между клиентом и сервером	425
Глава 7. Повторное использование разработанных кодов	433
7.1 Способы сохранения и повторного использования кодов	433
7.2 Создание и хранение шаблонов компонентов.	433
7.3 Создание новых компонентов и включение их в библиотеку.	435
7.3.1 Начало создания и установка компонента	435
7.3.2 Структура класса компонента	438
7.3.3 Задание свойств	439
7.3.4 Создание методов.	443
7.3.5 Создание событий	446
7.3.6 Автоматизация разработки классов в C++Builder 5	450

7.4 Депозитарий — хранилище форм, фреймов и проектов	454
7.5 Динамически присоединяемые библиотеки DLL	459
7.5.1 Назначение DLL	459
7.5.2 Статическое и динамическое присоединение DLL к приложению.	460
7.5.3 Создание DLL	461
7.6 Пакеты	466
7.6.1 Общее описание концепции пакетов.	466
7.6.2 Поддержка пакетов.	467
Глава 8. Разработка справочной системы (создание файлов .hlp).	471
8.1 Проектирование справочной системы	471
8.2 Создание файла тем справок	472
8.2.1 Написание текстов тем	472
8.2.2 Сноски.	473
8.2.3 Переходы	477
8.2.4 Макросы.	480
8.3 Компиляция и отладка	482
8.3.1 Создание файла Проекта справки	482
8.3.2 Компиляция и отладка справки	489
8.3.3 Файл содержания — .cnt	492
8.4 Особенности создания справки, работающей на любых версиях Windows.	493
8.4.1 Ограничения возможностей справки	493
8.4.2 Синтаксис файла Проекта.	494
ЧАСТЬ 3. Создание приложений для работы с базами данных.	497
Глава 9. Приложения для работы с локальными базами данных	499
9.1 Базы данных	499
9.1.1 Принципы построения баз данных.	499
9.1.2 Типы баз данных.	502
9.1.3 Технологии COM и CORBA	504
9.1.4 Организация связи с базами данных в C++Builder	506
9.2 Создание баз данных с помощью Database Desktop	507
9.2.1 Создание новой таблицы	507
9.2.2 Задание полей	508
9.2.3 Задание свойств таблицы	511
9.2.4 Завершение создания таблицы.	516
9.2.5 Изменение структуры и заполнение таблицы с помощью Database Desktop	517
9.3 Создание и редактирование псевдонимов баз данных, каталогов, драйверов	517
9.3.1 Автоматически создаваемые псевдонимы рабочего и частного каталогов.	517
9.3.2 Создание и просмотр псевдонимов баз данных в Database Desktop	518
9.3.3 Создание и просмотр псевдонимов драйверов и баз данных в BDE Administrator.	520
9.3.4 Создание и просмотр псевдонимов в SQL Explorer.	522
9.4 Обзор компонентов, используемых для связи с базами данных	523
9.5 Основные свойства компонента Table и простейшие приложения на его основе.	524
9.5.1 Установка связей между компонентами и базой данных, навигация по таблице.	524
9.5.2 Свойства полей.	527
9.5.3 Ограничения вводимых значений	529
9.5.4 Вычисляемые поля	530
9.5.5 Фильтрация данных	532
9.6 Использование словарей атрибутов полей	535

9.7 Некоторые компоненты визуализации и управления данными	538
9.8 Компонент Session	542
9.9 Компонент BatchMove	544
9.10 Приложения с несколькими связанными таблицами	545
9.10.1 Связь головной и вспомогательной таблиц	545
9.10.2 Поля просмотра (lookup fields)	547
9.11 Программирование работы с базами данных	548
9.11.1 Состояние набора данных	548
9.11.2 Пересылка записи в базу данных	549
9.11.3 Кэширование изменений	551
9.11.4 Доступ к полям	552
9.11.5 Методы навигации	554
9.11.6 Поиск записей	555
9.11.7 Методы установки диапазона допустимых значений	558
9.11.8 Методы модификации таблиц	559
9.11.9 Модули данных	560
9.12 Пример программирования работы с базой данных	565
Глава 10. Создание приложений для работы с базами данных в сети	571
10.1 Основы языка SQL и его использование в приложениях	571
10.1.1 Общие сведения	571
10.1.2 Оператор выбора Select	572
10.1.3 Операции с записями	578
10.1.4 Операции с таблицами	579
10.1.5 Операции с индексами	580
10.1.6 Компонент Query	581
10.1.7 Пример формирования произвольных запросов SQL	591
10.2 Работа с базами данных в сети	594
10.2.1 Транзакции и проблемы многопользовательского режима работы	594
10.2.2 Управление транзакциями, компонент Database	595
10.2.3 Работа с SQL Monitor	598
10.2.4 Управление доступом	599
10.3 InterBase — работа на платформе клиент/сервер	600
10.3.1 Общие сведения	600
10.3.2 Программа Server Manager	600
10.3.3 Windows ISQL	601
10.3.4 Обзоры — Views	605
10.3.5 Хранимые на сервере процедуры	606
10.4 Доступ к базам данных через Microsoft ActiveX Data Objects (ADO)	610
10.4.1 Соотношение между компонентами BDE и ADO	610
10.4.2 Задание соединения компонентов ADO с базой данных	612
10.4.3 Соединение с помощью компонента ADOConnection, управление транзакциями	615
10.4.4 Обзор компонентов наборов данных	617
10.5 Доступ к InterBase через InterBase Express (IBX)	621
10.5.1 Технология InterBase Express (IBX)	621
10.5.2 Компоненты IBDatabase и IBTransaction	623
10.5.3 Компоненты наборов данных IBTable, IBQuery, IBStoredProc	625
Глава 11. Обработка и документирование данных	627
11.1 Многомерный анализ данных — компоненты Decision Cube	627
11.1.1 Настройка компонентов приложения	627
11.1.2 Управление выполняющимся приложением	631
11.1.3 Компонент DecisionPivot	633
11.1.4 Компонент DecisionGraph	634
11.2 Создание отчетов	635
11.3 Использование серверов COM для документирования данных	642

ЧАСТЬ 4. Справочные сведения	651
Глава 12. Справочные данные по языку C++	653
12.1 Синтаксис языка	653
12.2 Директивы препроцессора	654
12.2.1 Директива #include	654
12.2.2 Директивы препроцессора #define и #undef.	655
12.2.4 Директивы #error, #line, #pragma	661
12.2.5 Операции препроцессора # и ##	662
12.3 Константы	662
12.3.1 Неименованные константы	662
12.3.2 Именованные константы	664
12.4 Переменные	665
12.4.1 Объявление переменных	665
12.4.2 Классы памяти	665
12.5 Функции	668
12.5.1 Объявление и описание функций	668
12.5.2 Передача параметров в функции по значению и по ссылке	672
12.5.3 Применение при передаче параметров спецификации const.	673
12.5.4 Параметры со значениями по умолчанию	675
12.5.5 Передача в функции переменного числа параметров	676
12.5.6 Встраиваемые функции inline	677
12.5.7 Перегрузка функций.	678
12.5.8 Шаблоны функций	679
12.6 Области видимости переменных и функций.	680
12.6.1 Правила, определяющие область видимости	680
12.6.2 Явное определение доступа с помощью объявлений namespace и using	682
12.7 Операции.	683
12.7.1 Общее описание	683
12.7.2 Арифметические операции	684
12.7.3 Операции присваивания, отличие присваивания от метода Assign	685
12.7.4 Операции отношения и эквивалентности	687
12.7.5 Логические операции	688
12.7.6 Поразрядные логические операции	688
12.7.7 Операция запятая (последование).	689
12.7.8 Условная операция (?:)	690
12.7.9 Операция sizeof	690
12.7.10 Операция typeid	691
12.7.11 Операции адресации (&) и косвенной адресации (*)	691
12.7.12 Операции разрешения области действия (::)	691
12.7.13 Операции доступа к элементам: точка (.) и стрелка (->)	692
12.7.14 Операции поместить в поток (<<) и взять из потока (>>)	692
12.7.15 Приоритет и ассоциативность операций	696
12.7.16 Перегрузка операций	697
12.8 Операторы	699
12.8.1 Операторы передачи управления	699
12.8.2 Операторы циклов	703
12.9 Динамическое распределение памяти.	708
12.10 Исключения.	712
12.10.1 Исключения и их стандартная обработка.	712
12.10.2 Способы защиты кодов зачистки — блоки try ... __finally и функции exit	713
12.10.3 Иерархия классов исключений VCL	716
12.10.4 Базовый класс исключений VCL Exception	720
12.10.5 Обработка исключений в блоках try ... catch	722
12.10.6 Преднамеренная генерация исключений	726
12.11 Сигналы	729

Глава 13. Типы данных в языке C++	731
13.1 Классификация типов данных, объявление типов	731
13.2 Приведение типов	734
13.3 Арифметические типы данных	735
13.4 Типы строк	737
13.4.1 Массивы символов	737
13.4.2 Тип строк AnsiString	740
13.5 Перечислимые типы	743
13.6 Множества	744
13.7 Указатели	747
13.8 Ссылки	750
13.9 Файлы и потоки	750
13.9.1 Файловый ввод/вывод с помощью компонентов	750
13.9.2 Файловый ввод/вывод с помощью потоков в стиле C	752
13.9.3 Файловый ввод/вывод с помощью потоков в стиле C++	762
13.10 Массивы	770
13.10.1 Одномерные массивы	770
13.10.2 Многомерные массивы	773
13.10.3 Операции с массивами, передача массивов как параметров	773
13.11 Структуры	775
13.11.1 Структуры в стиле C	775
13.11.2 Самоадресуемые структуры	777
13.11.3 Структуры в стиле C++	778
13.11.4 Битовые поля	779
13.12 Объединения	780
13.13 Классы	781
13.13.1 Объявление класса	781
13.13.2 Функции-элементы, дружественные функции, константные функции	784
13.13.3 Данные-элементы, статические данные, константные данные	786
13.13.4 Конструкторы и деструкторы	788
13.13.5 Наследование и полиморфизм, виртуальные функции, абстрактные классы	791
13.13.6 Особенности классов, наследующих классам библиотеки компонентов C++Builder	794
13.13.7 Шаблоны классов	797
Глава 14. Справочные данные по интегрированной среде разработки C++Builder	801
14.1 Структура меню C++Builder 5	801
14.1.1 Меню файлов File	801
14.1.2 Меню редактирования Edit	805
14.1.3 Меню поиска Search	808
14.1.4 Меню просмотра View	809
14.1.5 Меню проекта Project	811
14.1.6 Меню выполнения Run	814
14.1.7 Меню компонентов Component	816
14.1.8 Меню баз данных Database	817
14.1.9 Меню инструментов Tools	817
14.1.10 Меню справки Help	818
14.2 Настройка Интегрированной Среды Разработки C++Builder	819
14.2.1 Настройка инструментальной панели	819
14.2.2 Настройка палитры компонентов	820
14.2.3 Настройка Депозитария	820
14.2.4 Настройка меню Tools	821
14.2.5 Настройка Редактора Кода	824
14.2.6 Настройка Code Insight — Знатока Кода	828

14.2.7 Настройка Исследователя Классов ClassExplorer	829
14.2.8 Настройка отладчика	830
14.2.9 Настройка компилятора и компоновщика	834
14.2.10 Общие настройки среды.	848
14.3. Страницы библиотеки компонентов	851
14.3.1 Страница Standard.	851
14.3.2 Страница Additional	852
14.3.3 Страница Win32.	854
14.3.4 Страница System	855
14.3.5 Страница Data Access	856
14.3.6 Страница Data Controls	857
14.3.7 Страница ADO	858
14.3.8 Страница InterBase	859
14.3.9 Страница Decision Cube	860
14.3.10 Страница QReport	861
14.3.11 Страница Dialogs	862
14.3.12 Страница Win3.1	863
14.3.13 Страница Samples	864
14.3.14 Страница ActiveX	865
14.3.15 Страница Servers	866
Глава 15. Функции C, C++, библиотек C++Builder, API Windows	867
15.1 Справочные сведения общего характера	867
15.1.1 Коды клавиш	867
15.1.2 Некоторые объявленные константы C++Builder	870
15.1.3 Управляющие последовательности символов (escape-последовательности)	871
15.1.4 Форматы и типы, используемые при форматировании данных	873
15.1.5 Обработка ошибок времени выполнения, диагностика	884
15.2 Математические функции	888
15.3 Преобразование типов данных	899
15.3.1 Функции взаимного преобразования чисел и строк	899
15.3.2 Функции преобразования дат и времени	905
15.3.3 Функции преобразования типов	911
15.4 Строки и символы	912
15.4.1 Функции обработки символов	912
15.4.2 Функции обработки строк	915
15.5 Потоки и файлы	925
15.5.1 Атрибуты и флаги файлов, стандартные файлы	925
15.5.2 Управление потоками и файлами, описываемыми структурами FILE	928
15.5.3 Управление потоками и файлами, связанными с дескрипторами	931
15.5.4 Функции ввода/вывода	935
15.5.5 Функции обработки имен файлов.	941
15.5.6 Управление каталогами и файлами на дисках	944
15.6 Управление процессами	955
15.6.1 Функции управления текущим процессом.	955
15.6.2 Функции выполнения порождаемых процессов exes... и spawn...	958
15.6.3 Функции API Windows для запуска из приложения внешних программ	965
15.7 Функции различного назначения	971
15.7.1 Функции динамического распределения памяти	971
15.7.2 Функции вызова диалоговых окон с сообщениями	976
15.7.3 Функции воспроизведения звуков	988
15.7.4 Некоторые вспомогательные функции C++ и C++Builder.	992
15.7.5 Некоторые вспомогательные функции API Windows	997
15.8 Сообщения Windows	1000
15.8.1 Некоторые функции, константы и типы API Windows, используемые при работе с сообщениями	1000
15.8.2 Некоторые сообщения Windows	1003

Глава 16. Свойства, методы, события, типы, классы	1007
16.1 Свойства	1007
Action	1007
Align	1007
Anchors	1008
AutoMerge	1010
AutoSelect	1010
AutoSize	1010
Bitmap	1010
BoundsRect	1011
Break	1013
Brush	1013
Canvas	1014
Capacity	1014
Caption	1015
Charset	1016
ClientHeight	1017
ClientOrigin	1017
ClientRect	1018
ClientWidth	1018
ClipRect	1019
Color	1019
ComponentCount	1021
ComponentIndex	1021
Components	1021
Constraints	1022
ControlCount	1022
Controls	1023
ControlState	1024
ControlStyle	1025
CopyMode	1026
Count	1028
Ctl3D	1029
Cursor	1029
DesktopFont	1030
DockOrientation	1030
DragCursor	1031
DragKind	1031
DragMode	1031
DrawingStyle	1031
Enabled	1032
Font	1032
GroupIndex — свойство разделов меню	1033
Handle	1034
Height — свойство компонента	1034
Height — свойство шрифта	1035
HelpContext	1035
Hint	1035
HostDockSite	1036
ImageIndex	1037
Items — свойство класса TList	1037
Left	1038
List	1038
Mode — свойство TPen	1039
Name	1039
Parent	1040
ParentColor	1040
ParentCtl3D	1040
ParentFont	1041
ParentShowHint	1041
Pen	1042
PenPos	1042
Pitch	1042
Pixels	1043
PopupMenu	1043
ShortCut	1043
ShowHint	1044
Showing	1044
Size	1044
Style — свойство TPen	1045
Style — свойство TBrush	1045
Style — свойство TFont	1046
TabOrder	1046
TabStop	1047
Tag	1047

Text	1048
TextFlags	1048
Top	1049
TransparentColor	1049
TransparentMode	1049
Visible	1049
Width	1051
WindowText	1051
16.2 Методы	1051
Add	1051
Assign — метод графических объектов	1052
Assign — метод копирования объектов	1053
BeginDrag	1054
BringToFront	1054
BrushCopy	1056
CanFocus	1056
ChangeScale	1057
Chord	1057
ClassName	1058
Clear	1058
ClientToScreen	1059
ContainsControl	1059
ControlAtPos	1059
CopyRect	1060
DblClick	1060
Delete	1060
DisableAlign	1061
Dormant	1061
Draw	1062
DrawFocusRect	1062
Ellipse	1063
EnableAlign	1064
Exchange	1064
Expand	1065
FillRect	1065
FindNextControl	1066
FloodFill	1067
Focused	1067
FrameRect	1068
Free	1068
GetTabOrderList	1068
HandleAllocated	1069
HandleNeeded	1069
Hide	1069
IndexOf	1069
Insert	1070
Invalidate	1070
LineTo	1071
LoadFromClipboardFormat	1071
LoadFromFile — метод графических объектов	1072
LoadFromResourceID	1072
LoadFromResourceName	1072
LoadFromStream	1073
Lock	1073
Move	1073
MoveTo	1074
MouseCapture	1074
Pie	1074
PolyBezier и PolyBezierTo	1075
Polygon	1076
Polyline	1076
Realign	1077
Rectangle	1077
Refresh	1077
Remove	1077
Repaint	1078
ReplaceDockedControl	1078
RoundRect	1079
SaveToClipboardFormat	1079
SaveToFile — метод графических объектов	1080
SaveToStream	1081
ScaleBy	1081
ScaleControls	1082
ScreenToClient	1082
ScrollBy	1083
SelectFirst	1083

SelectNext	1084
SendCancelMode	1084
SendToBack	1084
SetBounds	1085
SetChildOrder	1085
SetFocus	1086
SetZOrder	1086
Show	1086
StretchDraw	1087
TextExtent	1087
TextHeight	1088
TextOut	1088
TextRect	1089
TextWidth	1090
TryLock	1090
Unlock	1090
Update	1091
16.3 События.	1091
OnChange — событие класса TCanvas	1091
OnChange — событие класса TGraphicsObject	1091
OnChangeing	1092
OnClick	1092
OnCreate	1093
OnDbiClick	1093
OnDragDrop	1093
OnDragOver	1095
OnEndDrag	1096
OnEnter	1096
OnExit	1097
OnKeyDown	1097
OnKeyPress	1098
OnKeyUp	1099
OnMouseDown и OnMouseUp	1100
OnMouseMove	1101
OnMouseUp	1102
OnPaint	1102
OnProgress	1102
OnStartDrag	1103
16.4 Некоторые базовые классы и типы	1104
Иерархия некоторых базовых классов библиотеки компонентов	1104
AnsiString — тип строк	1105
Set — шаблон класса	1109
TBitmap — класс	1110
TBrush — тип	1112
TCanvas — класс	1113
TColor — тип	1115
TComponent — базовый класс компонентов	1116
TControl — базовый класс визуальных компонентов	1118
TControlState — тип	1124
TControlStyle — тип	1124
TCursor — тип	1124
TCustomEdit — базовый класс окон редактирования	1124
TDragMode — тип	1126
TFont — тип	1126
TGraphic — базовый класс графических объектов	1128
TIcon — класс	1129
TImageList — компонент и класс	1131
TList — класс	1132
TMetafile — класс	1134
TObject — базовый класс всех объектов	1136
TPen — тип	1137
TPersistent — базовый класс объектов, участвующих в операциях с потоками	1137
TPicture — класс	1138
TPoint — тип	1140
TRect — тип	1140
TStringFloatFormat — тип	1141
TStringList — класс	1142
TStrings — класс	1143
TWinControl — базовый класс оконных компонентов	1145
16.5 Предметный указатель разделов книги, содержащих описания компонентов библиотеки VCL.	1149
Литература	1152

От автора

Для кого и о чем эта книга

Книга посвящена новой версии системы визуального объектно-ориентированного программирования C++Builder. Версия C++Builder 5 — превосходный инструмент, с помощью которого и начинающий пользователь, и программист-профессионал могут создавать одинаково профессионально выглядящий интерфейс пользователя к прикладным программам самых различных классов. Кроме того C++Builder позволяет работать с любыми базами данных, создавать прикладные программы для работы с Интернет и многое-многое другое. Так что недаром эта система пользуется широкой популярностью.

Впрочем, пока популярность C++Builder уступает популярности его родной сестры Delphi — разработанной той же фирмой Borland. Но мне кажется, что это явление временное. Язык C++, лежащий в основе C++Builder, более мощный, чем Object Pascal, на котором построена Delphi. И библиотеки функций C++ намного обширнее библиотек Object Pascal. Поэтому то, что в C++Builder делается легко и естественно, в Delphi в ряде случаев требует значительно больших усилий и получается не столь эффективно. Правда, это касается только весьма сложных приложений. Большинство же прикладных задач с равным успехом могут решаться и средствами Delphi, и средствами C++Builder.

Меньшая популярность C++Builder по сравнению с Delphi объясняется, на мой взгляд, большей сложностью (неизбежной при большой мощности) языка C++. Но думаю, что это временное препятствие. Уже сейчас в ряде вузов начинают изучать C и C++ вместо традиционного языка Pascal. Так что для нового поколения разработчиков C++Builder может оказаться более естественным, чем Delphi. Да и наиболее серьезные разработчики старшего поколения тоже на ты с C++. Все это вселяет надежду, что в недалеком будущем популярность C++Builder догонит, а может быть и обгонит популярность Delphi.

При написании данной книги я старался сделать ее полезной для читателей различных категорий — от начинающих (но, все-таки, хотя бы поверхностно знакомых с каким-нибудь языком программирования) до опытных профессионалов. Поэтому я отказался от стиля учебника с его последовательным изложением материала от простого к сложному. Каждая глава книги является законченным изложением того или иного вопроса. Я с уважением отношусь к своим потенциальным читателям и считаю, что они сами смогут регулировать последовательность и темп своего изучения C++Builder или своей работы с этой системой. К тому же, я уверен, что овладеть какой-то системой можно только работая с ней, решая какую-то свою (не учебную) конкретную задачу. И последовательность изучения тех или иных вопросов определяется требованиями этой конкретной задачи. Поэтому я считал основной задачей данной книги — дать побольше конкретной информации по всем затронутым в ней вопросам и расположить эту информацию в такой последовательности, чтобы ее легко было отыскать. Тогда и начинающий, и опытный пользователь сможет в любой момент найти то, что ему нужно сейчас для работы. Надо отметить, что опытному пользователю тоже нужна подобная книга, поскольку никто не в состоянии держать в голове все сведения по многим десяткам компонентов C++Builder, по синтаксису и параметрам тех или иных функций и процедур языка программирования.

При написании этой книги я пытался дать читателю как можно больше конкретной информации. Насколько это удалось — судить читателю. А я пытался ре-

шить эту задачу следующим образом. Во-первых, изложение в книге предельно сжатое, без лирических отступлений. Во-вторых, наряду со связным изложением отдельных вопросов в книге имеется большая справочная часть, содержащая конкретную информацию по функциям и языку C++, по свойствам, методам, событиям компонентов. Это позволило не нарушать ход изложения и в то же время дать большой фактический материал, не отсылая читателя то и дело к справке C++Builder. В третьих, отдельные главы книги написаны так, как если бы это были специальные книги по соответствующим вопросам. Я стремился к полноценному, законченному изложению каждой темы. А многочисленные ссылки на другие главы книги позволили избежать дублирования материала. В четвертых, и я, и издательство сделали все возможное, чтобы вместить в разумные габариты максимум материала. В частности, пришлось пойти на достаточно мелкий шрифт. Может быть не всем читателям это по душе, но если бы не такой шрифт, книга была бы еще в полтора раза толще (и дороже).

Отличие от книг серии «Все о C++Builder» и «Программирование в C++Builder 4»

Для читателей, которые видели, а, может быть, и имеют какие-то другие мои книги, посвященные C++Builder, необходимо пояснить соотношение тех книг и этой. Начну с книги — «Программирование в C++Builder 4». Структура данной книги осталась прежней (хотя и добавилась одна глава), так как, вроде бы, со стороны читателей нареканий на этот счет не было. Материал, конечно, частично повторяется, хотя практически во всех главах добавлено новое и устранены некоторые замеченные недочеты прежней книги. Но добавлено и очень много совершенно нового материала, в частности:

- интернационализация разрабатываемых прикладных программ
- развертывание, установка и настройка прикладных программ
- новые альтернативные технологии доступа к базам данных — Microsoft ActiveX Data Objects и InterBase Express
- многомерный анализ данных — компоненты Decision Cube
- компоненты — серверы COM
- компоненты страницы ActiveX
- новые компоненты C++Builder 5 — фреймы, диспетчер событий приложения и ряд других
- новые возможности среды проектирования C++Builder 5 и новый инструментарий, встроенный в эту среду: Проектировщик Модулей Данных, автоматизация задания в классах новых свойств, методов, событий, сообщений и многое другое

Так как необходимо было вписываться в тот же уже весьма большой объем книги, то пришлось в ряде случаев уплотнить изложение и кое-какой второстепенный материал убрать. Надеюсь, что это сделано не в ущерб качеству.

Теперь о соотношении данной книги с книгами [1] — [8] серии «Все о C++Builder» (список этих книг дан в конце). Серия была задумана как способ дать читателю возможность выборочно приобретать литературу только по тем вопросам, которые его интересуют, причем в более углубленном, более подробном изложении и с большим числом примеров. Поэтому, хотя многое в первых восьми книгах серии является дословным повторением материала данной книги, они содержат немало дополнительной информации, примеров, справочных данных. В первую очередь это относится к книгам [2] и [3], посвященным компонентам библиотеки C++Builder, и к книгам [7] [8], посвященным работе с базами данных. Так что я мог бы взять на себя смелость рекомендовать тем читателям, которые хотят получить больше информации по этим вопросам, познакомиться с указанными

книгами. А последующие книги серии «Все о C++Builder», которые намечено подготовить, будут содержать материал, который в данной книге вообще отсутствует или только бегло затронут.

Что вы найдете в этой книге

Итак, что же вы можете найти в этой книге.

Книга состоит из четырех частей. *Первая часть* — вводная. В ней излагаются основы объектно-ориентированного визуального программирования (глава 1), методика работы с интегрированной средой разработки C++Builder (глава 2) и рассматриваются наиболее часто используемые компоненты из библиотеки C++Builder (глава 3). Первые две главы построены как начальный курс обучения работе с C++Builder. Третья глава дает справочный материал по большинству компонентов (остальные компоненты рассмотрены в других главах). Пользователь C++Builder, как и любой другой большой системы, обычно использует только часть ее возможностей, только свои любимые компоненты, которые он изучил. Поэтому, вероятно, даже тем, кто уже работал с C++Builder, будет очень полезно посмотреть материал главы 3, чтобы обогатить свою палитру используемых компонентов.

Вторая и третья части книги — методические. *Вторая часть*, включающая главы с 4 по 8, посвящена различным аспектам построения прикладных программ для Windows. В каждой из этих глав идет последовательное изложение материала, сопровождаемое многочисленными примерами. Фактически, каждую главу можно рассматривать как раздел учебного курса по соответствующему вопросу. Но, поскольку все аспекты построения приложений в C++Builder тесно связаны друг с другом, пришлось выбирать один из двух возможных вариантов построения книги: или придерживаться строгой последовательности изложения (в каждой главе использовать только то, о чем было сказано в предыдущих главах), что неминуемо ведет к потере полноты изложения, или излагать каждый вопрос полностью, но тогда допускать при необходимости немало ссылок на материалы, изложенные в последующих главах. Учебники строятся по первому принципу и это является причиной того, что большинство книг по C++Builder грешат неполнотой: рассматриваются основы какой-то темы, но многие важнейшие ее аспекты опускаются, поскольку для их изложения нужны сведения из еще не изученного раздела. В этой книге принят второй подход: каждая глава излагает соответствующую тему полностью и содержит ссылки на разделы других глав, в которых читатель может прояснить для себя какие-то непонятные ему вопросы. Таким образом, каждая глава может рассматриваться как законченный методический и справочный материал по соответствующей теме. Благодаря этому читатель может изучать главы в той последовательности, какая нужна ему для его конкретной работы, опуская главы, не представляющие для него в данный момент интереса. Исключение, может быть, представляет только глава 4, в которой рассматриваются наиболее общие вопросы построения прикладных программ для Windows и которую, вероятно, надо прочитать (может быть, с некоторыми купюрами), независимо от конкретных пристрастий читателя к той или иной тематике.

Глава 5 посвящена вопросам построения приложений, использующих графику и средства мультимедиа. В главе 6 рассматриваются различные аспекты взаимодействия приложения с внешними программами. Эти вопросы нередко излагаются в литературе по C++Builder очень бегло. Однако, опыт показывает, что часто к помощи C++Builder обращаются именно для построения пользовательского интерфейса к уже имеющимся программам, разработанным ранее для Windows или DOS. Поэтому взаимодействие с этими готовыми программами хотя бы просто на уровне вызова их выполняемых модулей для многих разработчиков является очень важным вопросом. Впрочем, в главе 6 рассматривается и методика более тонкого взаимодействия приложений: использование сообщений Windows, техно-

логии OLE и DDE и серверы COM — новый класс компонентов в C++Builder 5. Глава 7 рассматривает различные способы повторного использования разработанных вами кодов: построение шаблонов и создание новых компонентов, разработку иерархии форм и фреймов, библиотек DLL, пакетов. Владение различными способами повторного использования кодов может экономить массу времени при разработке серии приложений и способствует тому, что с каждым новым приложением ваши затраты на разработку будут становиться все меньше и меньше. В главе 8 детально излагается методика построения справочных файлов Windows. Это не связано непосредственно с C++Builder и поэтому, если этот вопрос и затрагивается в книгах о C++Builder, то очень поверхностно. Но в настоящее время трудно представить себе серьезную прикладную программу для Windows, не содержащую встроенной в нее контекстной справочной системы. Поэтому построению таких систем в этой книге уделено должное внимание.

Часть третья книги, содержащая главы 9, 10 и 11, посвящена созданию приложений для работы с базами данных. Создание таких приложений — наиболее мощная сторона C++Builder. Изложение в этих главах ведется последовательно и сопровождается многочисленными примерами. Эта часть книги может рассматриваться как самостоятельный курс по созданию в среде C++Builder приложений, работающих с базами данных, и по разработке самих баз данных с помощью C++Builder.

Четвертая часть книги, наиболее объемная и включающая главы с 12 по 16, содержит справочные материалы. Они вынесены в отдельную часть и в отдельные главы, чтобы не загромождать техническими деталями изложение в предыдущих частях книги. К тому же, это облегчает читателю поиск интересующих его конкретных сведений. В главах 12 и 13 даются справочные сведения по языку программирования C++. При этом в главе 12 рассматривается синтаксис языка, объявления переменных, функций, общие вопросы построения программы. А глава 13 посвящена справочным сведениям о типах в C++, начиная от простых, и кончая такими, как массивы, структуры и классы. В главе 14 приведены справочные материалы по интегрированной среде разработки C++Builder: описания команд меню, палитры компонентов, настроек среды. Глава 15 содержит справочные данные по функциям C++, C++Builder и API Windows. Здесь приводятся сведения по более чем 570 функциям. В главе 16 в алфавитном порядке приведены сведения о свойствах, методах, событиях компонентов и классов C++Builder и сведения о некоторых базовых классах и типах C++Builder. В последнем разделе главы дается алфавитный перечень компонентов C++Builder со ссылками на разделы книги, в которых эти компоненты подробно рассмотрены. Этот раздел полезен для быстрого поиска методической и справочной информации по тому или иному компоненту.

Конечно, перечень функций, элементов C++Builder, их свойств и методов в главах 15 и 16 не полный, хотя и внушительный: в них рассмотрено свыше 570 функций и около 400 методов, свойств, событий различных классов (некоторые подробно, некоторые бегло). Большого объема сведений эта книга просто не выдержала бы, так как попытка охватить весь справочный материал привела бы к тому, что объем глав 15 и 16 в два раза превысил бы объем всей этой книги. Но большинство функций, свойств, методов, событий, упоминаемых в первых частях книги и нуждающихся в более детальном рассмотрении, в этих главах приведены. И к этому добавлено многое, о чем в предыдущих главах не говорилось.

Прилагаемый к книге диск содержит многие из примеров и баз данных, рассмотренных в книге. Они отличаются от демонстрационных примеров, распространяемых с C++Builder, большей прозрачностью кодов (в них намеренно пропущены некоторые усложнения, затрудняющие чтение законченных реальных приложений) и подробным описанием в виде отдельных текстов и комментариев.

На диске имеется также начальный вариант файла справки по C++Builder и C++ на русском языке. Конечно, это не полная справка по C++Builder, а только ее нулевая версия. Справку можно встроить в интегрированную среду разработки

C++Builder и использовать в своей текущей работе. Работа над справкой продолжается и, если решатся некоторые чисто технические вопросы, то первая полноценная версия справки выйдет в скором времени в серии «Все о C++Builder».

Чего вы не найдете в этой книге

Теперь, рассмотрев то, что вы можете найти в этой книге, полезно сразу четко определить, чего вы в ней не найдете. Должен честно оговориться, что эта книга, как и любая литература по C++Builder, не претендует на исчерпывающую полноту. Привести в одной книге подробные сведения по всем функциям языка, по всем классам, типам и программным составляющим постоянно развивающейся системы C++Builder не представляется возможным. Поэтому, отступая от традиции, по которой авторы просто тихо умалчивают о том, что не вошло в книгу, хочу сразу честно сориентировать читателя, чтобы он не тратил силы на поиск того, чего найти невозможно.

В книге не рассмотрена работа приложений C++Builder с Интернет и, значит, не рассматриваются соответствующие страницы библиотеки компонентов. Не рассматривается также страница *MIDAS* библиотеки компонентов, связанная с построением приложений для распределенных баз данных. Эта страница имеется только в вариантах C++Builder клиент/сервер и Enterprise. Соответственно, в книге обсуждаются приложения, работающие со структурами баз данных от локальных до клиент/сервер, и не затрагиваются многоуровневые базы данных. В книге крайне конспективно рассматриваются вопросы, связанные с использованием ActiveX и CORBA (Common Object Request Broker Architecture — стандарт построения приложений с распределенными объектами). Из возможных видов прикладных программ, которые можно строить с помощью C++Builder 5, рассмотрены практически только программы для Windows. Консольные приложения WIN32 только упоминаются. Это, на мой взгляд, оправдано, поскольку именно в приложениях для Windows проявляется вся мощь C++Builder. А консольные приложения можно строить с тем же успехом и с помощью других инструментов.

Вот, пожалуй, и все. Перечисленные вынужденные умолчания связаны с естественными ограничениями на объем книги. Было решено лучше полностью излагать материал по существенным вопросам, чем сокращать это изложение ради беглого рассмотрения того, что осталось за кадром. На мой взгляд, того, что рассмотрено в книге, более чем достаточно для построения подавляющего большинства приложений. Ну а если читателю потребуются сведения по неосвещенным вопросам, ему придется обратиться к специальной литературе или работать со встроенной справкой C++Builder. Я надеюсь также, что эти пробелы будут в скором времени устранены книгами серии «Все о C++Builder». Кстати, приглашаю любого специалиста, владеющего материалом по вопросам, не освещенным в данной книге, стать автором соответствующей книги серии «Все о C++Builder». Присылайте свои предложения в адрес издательства.

Рекомендации по работе с книгой

Выше я уже написал, что доверяю способности читателя самому выбрать наиболее удобную ему последовательность изучения материала данной книги. Благо она построена так, что отдельные темы можно изучать независимо друг от друга и в любой последовательности. Но все же я возьму на себя смелость дать некоторые советы по изучению материала разными категориями читателей.

Тем, кто до чтения этой книги никогда не работал с C++Builder, вероятно, будет полезной следующая последовательность работы. Сначала надо прочитать материал небольшой главы 1, вводящей читателя в мир объектно-ориентированного программирования. Затем следует научиться работать в интегрированной среде разработки C++Builder с помощью последовательного изучения материалов гла-

вы 2 (ряд тонкостей, связанных с настройкой среды проектирования и с отладкой приложений, можно при этом пропустить, вернувшись к этому материалу позднее). Две первые главы книги построены как начальный курс обучения работе с C++Builder и не должны вызвать затруднений у тех, кто впервые установил эту систему на своем компьютере. Тем из начинающих, кто не знаком с языком программирования C++, можно рекомендовать по началу не отвлекаться на его изучение (некоторый минимум сведений о нем содержится в разделе 1.6 главы 1), а смело создавать по рекомендациям главы 2 и последующих глав свои первые приложения. Мой опыт преподавания C++Builder говорит, что это вполне возможно. Если что-то в кодах этих приложений вам будет не понятно, обращайтесь за разъяснениями к соответствующим разделам глав 12, 13 и 15. Более внимательное изучение этих глав целесообразно отнести к тому времени, когда вы уже хорошо освоитесь с C++Builder и вам захочется перейти к каким-то сложным приложениям, в которых потребуются использовать более глубокие аспекты языка C++.

После начального обучения по материалам глав 1 и 2 желательно последовательно изучить в главе 4 материал разделов 4.1 — 4.3 и раздела 4.5 («Формы»). По мере изучения этих разделов полезно обращаться к соответствующим разделам главы 3, чтобы получить дополнительную информацию об используемых в примерах компонентах.

Изучив все это, вы можете перестать считать себя начинающим пользователем и планировать дальнейшее знакомство с материалом других глав и с пропущенными разделами главы 4 исходя из того, что вам более всего нужно в вашей работе.

Теперь некоторые рекомендации более или менее опытным пользователям C++Builder. Можно, вероятно, не читать главу 1, но ряд разделов главы 2, связанных с различными инструментами среды C++Builder 5 и с их настройкой, просмотреть полезно. Жизнь показывает, что даже достаточно опытные пользователи используют далеко не все возможности, встроенные в C++Builder. Во всяком случае, работая над материалом главы 2, я открыл для себя ряд возможностей, о которых ранее не знал. А в среде разработки C++Builder 5 появилось немало нового инструментария, который очень полезно освоить.

Полезно также просмотреть материал главы 3, посвященной компонентам C++Builder общего назначения. Не говоря уже о новых компонентах, появившихся в C++Builder 5, о новых способах диспетчеризации событий, вы, вероятно все-го, и из прежних компонентов используете далеко не все, отдавая предпочтение наиболее любимым вами. Так что посмотреть сравнительный анализ различных компонентов будет в любом случае полезно.

В главе 4 полезно ознакомиться с новой технологией интернационализации приложений, введенной в Delphi 5, в главах 6 и 11 — с серверами COM — новым классом компонентов в Delphi 5, в главе 10 — с новыми альтернативными способами доступа к базам данных. А в остальном — изучайте то, что вам надо, и в любой последовательности. Опыт подскажет вам наиболее эффективный способ работы с материалом.

Благодарности

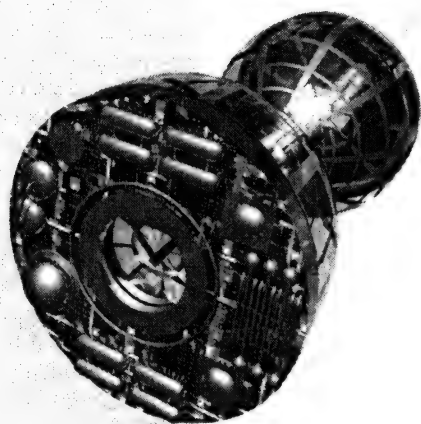
Благодарю издательство БИНОМ, инициировавшее написание книги, и представительство фирмы Borland в Москве, предоставившее возможность ознакомиться с C++Builder 5. Выражаю искреннюю благодарность Максиму Тагину, много сделавшему для разработки файла справки по C++Builder.

Благодарю и тебя, мой читатель, за интерес к этой книге. Я с удовольствием приму любые замечания и предложения по ее совершенствованию, поскольку планирую и далее заниматься той же тематикой. Кроме того, обрати внимание на содержащийся выше призыв к сотрудничеству в создании серии «Все о C++Builder». Поделись с другими своим опытом работы с C++Builder. Любые предложения будут с благодарностью приняты и рассмотрены редакцией.

Часть I

Объектно-ориентированное визуальное программирование

- Глава 1 Принципы объектно-ориентированного визуального программирования
- Глава 2 Система визуального объектно-ориентированного программирования C++Builder
- Глава 3 Обзор компонентов библиотеки C++Builder





Глава 1

Принципы объектно-ориентированного визуального программирования

1.1 Объектно-ориентированное программирование

Объектно-ориентированный подход уже давно стал естественным для любых прикладных программ Windows (далее на протяжении всей книги мы для краткости будем называть прикладные программы *приложениями*). Когда вы начинаете выполнять любую программу Windows, вы видите обычно диалоговое окно с меню, кнопками, индикаторами, выпадающими списками и т.п. Все это *объекты*. Сами по себе они не производят никаких действий, пока не получат какое-то *сообщение* от пользователя. После получения сообщения происходят какие-то действия (проводятся вычисления, выполняется какая-то программа, появляется новое диалоговое окно и т.п.). После этого опять всякая активность опять затухает, пока тот или иной объект не получит нового сообщения от пользователя или от другого объекта. Таким образом, объектно-ориентированная программа не имеет жесткого алгоритма работы. Она представляет собой систему объектов, каждый из которых может выполнять какие-то функции в ответ на полученное сообщение, в частности, может сам генерировать сообщения, на которые будут реагировать другие объекты. Взаимодействие пользователя с компьютерной программой — это тоже взаимодействие двух объектов — программы и человека, которые обмениваются друг с другом определенными сообщениями.

Попробуем разобраться с основным понятием объектно-ориентированного программирования — объектом. Для начала можно определить объект как некую совокупность данных и способов работы с ними. Данные можно рассматривать как поля записи. Это характеристики объекта. Пользователь и объекты программы должны, конечно, иметь возможность читать эти данные объекта, как-то их обрабатывать и записывать в объект новые значения.

Здесь важнейшее значение имеют принципы инкапсуляции и скрытия данных. Принцип скрытия данных заключается в том, что внешним объектам и пользователю прямой доступ к данным, как правило, запрещен. Делается это из двух соображений.

Во-первых, для надежного функционирования объекта надо поддерживать целостность и непротиворечивость его данных. Если не позаботиться об этом, то внешний объект или пользователь могут занести в объект такие неверные данные, что он начнет функционировать с ошибками.

Во-вторых, необходимо изолировать внешние объекты от особенностей внутренней реализации данных. Для внешних потребителей данных должен быть доступен только пользовательский интерфейс — описание того, какие имеются данные и функции и как их использовать. А внутренняя реализация — это дело разра-

ботчика объекта. При таком подходе разработчик может в любой момент модернизировать объект, изменить структуру хранения и форму представления данных, но, если при этом не затронут интерфейс, внешний потребитель этого даже не заметит. И, значит, во внешней программе и в поведении пользователя ничего не придется менять.

Чтобы выдержать принцип скрытия данных, в объекте обычно определяются процедуры и функции, обеспечивающие все необходимые операции с данными: их чтение, преобразование, запись. Эти функции и процедуры называются *методами* и через них происходит общение с данными объекта (рис. 1.1).

Рис. 1.1

Схема обращения к данным через методы объекта



Совокупность данных и методов их чтения и записи называется *свойством*. Со свойствами вы будете иметь дело на протяжении всей этой книги. Свойства можно устанавливать в процессе проектирования, их можно изменять программно во время выполнения вашей прикладной программы. Причем внешне это все выглядит так, как будто объект имеет какие-то данные, например, целые числа, которые можно прочитать, использовать в каких-то вычислениях, заложить в объект новые значения данных. В процессе проектирования вашего приложения с помощью C++Builder вы можете видеть значения некоторых из этих данных в окне Инспектора Объектов (см. главу 2), можете изменять эти значения. В действительности все обстоит иначе. Все общение с данными происходит через методы их чтения и записи. Это происходит и в процессе проектирования, когда среда проектирования C++Builder запускает в нужный момент эти методы, и в процессе выполнения приложения, поскольку компилятор C++Builder незримо для разработчика вставляет в нужных местах программы вызовы этих методов. Например, если в объекте имеется некоторое числовое свойство **A**, то вы можете написать оператор вида

```
A = A + 1;
```

который прибавляет к значению **A** единицу. Но в действительности реализация этого оператора во многих случаях выглядит сложнее. Компилятор преобразует этот простой код в вызов метода чтения свойства **A**, к возвращенному этим методом значению прибавляется единица, а затем вызывается метод записи свойства, который заносит полученный результат в данные объекта.

Помимо методов, работающих с отдельными данными, в объекте имеются методы, работающие со всей их совокупностью, меняющие их структуру. Таким образом, объект является совокупностью свойств и методов. Но это пока нельзя считать законченным определением объекта, поскольку прежде, чем дать полное определение, надо еще рассмотреть взаимодействие объектов друг с другом.

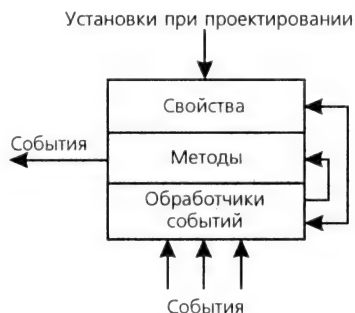
Средой взаимодействия объектов (как бы силовым полем, в котором существуют объекты) являются *сообщения*, генерируемые в результате различных *событий*. События наступают, прежде всего, вследствие действий пользователя — перемещения курсора мыши, нажатия кнопок мыши или клавиш клавиатуры. Но события могут наступать и в результате работы самих объектов. В каждом объекте определено множество событий, на которые он может реагировать. В конкретных экземплярах объекта могут быть определены *обработчики* каких-то из этих событий, которые и определяют реакцию данного экземпляра объекта. В обработчиках анализируется сгенерированное событие и предпринимаются те или иные действия: изменение свойств объектов, выполнение каких-то методов, генерация новых

событий. К написанию этих обработчиков, часто весьма простых, и сводится, как будет видно далее, основное программирование при разработке *графического интерфейса пользователя* с помощью C++Builder.

Теперь можно окончательно определить объект как совокупность свойств и методов, а также событий, на которые он может реагировать. Условно это изображено на рис. 1.2. Внешнее управление объектом осуществляется через обработчики событий. Эти обработчики обращаются к методам и свойствам объекта. Начальные значения данных объекта могут задаваться также в процессе проектирования установкой различных свойств. В результате выполнения методов объекта могут генерироваться новые события, воспринимаемые другими объектами программы или пользователем.

Рис. 1.2

Схема организации объекта



Описанная ранее структура программы в виде некоторой фиксированной совокупности объектов не является полной. Чаще всего сложная программа — это не просто какая-то предопределенная совокупность объектов. В процессе работы объекты могут создаваться и уничтожаться. Таким образом, структура программы является *динамическим образованием*, меняющимся в процессе выполнения. Основная цель создания и уничтожения объектов — экономия ресурсов компьютера и, прежде всего, памяти. Несмотря на бурное развитие вычислительной техники, память, наверное, всегда будет лимитировать возможности сложных приложений. Это связано с тем, что сложность программных проектов растет теми же, если не более быстрыми, темпами, что и техническое обеспечение. Поэтому от объектов, которые не нужны на данной стадии выполнения программы, нужно освобождаться. При этом освобождаются и выделенные им области памяти, которые могут использоваться вновь создаваемыми объектами. Простой пример этого — окно-заставка с логотипом, появляющееся при запуске многих приложений. После начала реального выполнения приложения эта заставка исчезает с экрана и никогда больше не появляется в данном сеансе работы. Было бы варварством не уничтожить этот объект и не освободить занимаемую им память для более продуктивного использования.

С целью организации динамического распределения памяти во все объекты заложены методы их создания — *конструкторы* и уничтожения — *деструкторы*. Конструкторы объектов, которые изначально должны присутствовать в приложении (прикладной программе), срабатывают при запуске программы. Деструкторы всех объектов, имеющих в данный момент в приложении, срабатывают при завершении его работы. Но нередко и в процессе выполнения различные новые объекты (например, новые окна документов) динамически создаются и уничтожаются с помощью их конструкторов и деструкторов.

Включать объекты в свою программу можно двумя способами: *вручную* включать в нее соответствующие операторы (это приходится делать не очень часто) или путем *визуального программирования*, используя заготовки — *компоненты*.

1.2 Основы визуального программирования интерфейса

Сколько существует программирование, столько существуют в нем и тупики, в которые оно постоянно попадает и из которых в конце концов доблестно выходит. Один из таких тупиков или кризисов не так давно был связан с разработкой графического интерфейса пользователя. Программирование вручную всяких привычных пользователю окон, кнопок, меню, обработка событий мыши и клавиатуры, включение в программы изображений и звука требовало все больше и больше времени программиста. В ряде случаев весь этот сервис начинал занимать до 80-90% объема программных кодов. Причем весь этот труд нередко пропадал почти впустую, поскольку через год — другой менялся общепринятый стиль графического интерфейса и все приходилось начинать заново.

Выход из этой ситуации обозначился благодаря двум подходам. Первый из них — стандартизация многих функций интерфейса, благодаря чему появилась возможность использовать библиотеки, имеющиеся, например, в Windows. В итоге при смене стиля графического интерфейса (например, при переходе от Windows 3.x к Windows 95) приложения смогли автоматически приспособливаться к новой системе без какого-либо перепрограммирования. На этом пути создались прекрасные условия для решения одной из важнейших задач совершенствования техники программирования — *повторного использования кодов*. Однажды разработанные вами формы, компоненты, функции могли быть впоследствии неоднократно использованы вами или другими программистами для решения их задач. Каждый программист получил доступ к наработкам других программистов и к огромным библиотекам, созданным различными фирмами. Причем была обеспечена совместимость программного обеспечения, разработанного на разных алгоритмических языках.

Вторым революционным шагом, кардинально облегчившим жизнь программистов, явилось появление визуального программирования, возникшего в Visual Basic и нашедшего блестящее воплощение в системах Delphi и C++Builder фирмы Borland. Это явилось решающим шагом в развитии так называемой CASE-технологии (Computer Aided Software Engineering — автоматизированное проектирование программного обеспечения).

Визуальное программирование позволило свести проектирование пользовательского интерфейса к простым и наглядным процедурам, которые дают возможность за минуты или часы сделать то, на что ранее уходили месяцы работы. В современном виде в C++Builder это выглядит так.

Вы работаете в Интегрированной Среде Разработки (ИСР или Integrated development environment — IDE) C++Builder. Среда предоставляет вам *формы* (в приложении их может быть несколько), на которых размещаются компоненты. Обычно это оконная форма, хотя могут быть и невидимые формы. На форму с помощью мыши переносятся и размещаются пиктограммы компонентов, имеющихся в библиотеках C++Builder. С помощью простых манипуляций вы можете изменять размеры и расположение этих компонентов. При этом вы все время в процессе проектирования видите результат — изображение формы и расположенных на ней компонентов. Вам не надо мучиться, многократно запуская приложение и выбирая наиболее удачные размеры окна и компонентов. Результаты проектирования вы видите, даже не компилируя программу, немедленно после выполнения какой-то операции с помощью мыши.

Но достоинства визуального программирования не сводятся к этому. Самое главное заключается в том, что во время проектирования формы и размещения на ней компонентов C++Builder автоматически формирует коды программы, включая в нее соответствующие фрагменты, описывающие данный компонент. А затем

в соответствующих диалоговых окнах пользователь может изменить заданные по умолчанию значения каких-то свойств этих компонентов и, при необходимости, написать обработчики каких-то событий. То есть проектирование сводится, фактически, к размещению компонентов на форме, заданию некоторых их свойств и написанию, при необходимости, обработчиков событий.

Компоненты могут быть *визуальные*, видимые при работе приложения, и *невизуальные*, выполняющие те или иные служебные функции. Визуальные компоненты сразу видны на экране в процессе проектирования в таком же виде, в каком их увидит пользователь во время выполнения приложения. Это позволяет очень легко выбрать место их расположения и их дизайн — форму, размер, оформление, текст, цвет и т.д. Невизуальные компоненты видны на форме в процессе проектирования в виде пиктограмм, но пользователю во время выполнения они не видны, хотя и выполняют для него за кадром весьма полезную работу.

В библиотеки визуальных компонентов C++Builder включено множество типов компонентов и их номенклатура очень быстро расширяется от версии к версии. Имеющегося уже сейчас вполне достаточно, чтобы построить практически любое самое замысловатое приложение, не прибегая к созданию новых компонентов. При этом даже неопытный программист, делающий свои первые шаги на этом поприще, может создавать приложения, которые выглядят совершенно профессионально.

1.3 Предварительные сведения о классах и наследовании

Типы объектов и, в частности, компонентов библиотек C++Builder оформляются в виде *классов*. Классы — это типы, определяемые пользователем. В классах описываются свойства объекта, его методы и события, на которые он может реагировать. Язык программирования C++ предусматривает только инструментарий создания классов. А сами классы создаются разработчиками программного обеспечения. Впрочем, создатели C++Builder уже разработали для вас множество очень полезных классов и включили их в библиотеки системы. Этими классами вы пользуетесь при работе в Интегрированной Среде Проектирования. Конечно, это несколько не мешает создавать вам свои новые классы.

Если бы при создании нового класса вам пришлось все начинать с нуля, то эффективность этого занятия была бы под большим сомнением. Да и разработчики C++Builder вряд ли создали бы в этом случае такое множество классов. Действительно, представьте себе, что при разработке нового компонента, например, какой-нибудь новой кнопки, вам пришлось бы создавать все: рисовать ее изображение, описывать все свойства, определяющие ее место расположения, размеры, надписи и картинки на ее поверхности, цвет, шрифты, описывать методы, реализующие ее поведение — изменение размеров, видимость, реакции на сообщения, поступающие от клавиатуры и мыши. Вероятно, представив себе все это, вы отказались бы от разработки новой кнопки.

К счастью, в действительности все обстоит гораздо проще, благодаря одному важному свойству классов — *наследованию*. Новый класс может наследовать свойства, методы, события своего *родительского класса*, т.е. того класса, на основе которого он создается. Например, при создании новой кнопки можно взять за основу один из уже разработанных классов кнопок и только добавить к нему какие-то новые свойства или отменить какие-то свойства и методы родительского класса.

1.4 Системы быстрой разработки приложений

Благодаря визуальному объектно-ориентированному программированию была создана технология, получившая название *быстрая разработка приложений*, по-английски *RAD* — Rapid Application Development. Эта технология характерна для нового поколения систем программирования, к которому относится и C++Builder.

Первой ласточкой в этом новом мире более простого и наглядного интерфейса была среда Visual Basic. Она сформировала новый стиль взаимодействия разработчика программы с компьютером, позволяя наглядно конструировать пользовательский интерфейс с помощью мыши, а не обычным для прежних времен путем: написанием кодов, их трансляцией и выполнением программы, после чего только и можно было посмотреть, как же это выглядит на экране.

Хотя Visual Basic нашел широкий спрос и помог открыть мир программирования для людей, не слишком в нем искушенных, он не свободен от многих проблем. Главные из них — низкая производительность разрабатываемых приложений при их выполнении, недостаточная строгость и объектная ориентированность языка, способствующая скорее быстрой разработке поделок, а не созданию мощных эффективных приложений, а также ряд других недостатков.

Системы Delphi и C++Builder — это следующий шаг в развитии среды быстрой разработки приложений. Они исправляют многие дефекты, обнаруженные в Visual Basic. Разработчики этих систем создали инструменты, которые на первый взгляд выглядят похожими на среду Visual Basic, хотя в действительности они заметно лучше.

Интегрированная среда разработки в Delphi и C++Builder выглядит одинаково. Весь пользовательский интерфейс, все библиотеки, все приемы работы с этими системами практически одинаковы. Если быть более точным, то они различаются только в силу разного времени выпуска соответствующих версий. Версии C++Builder выпускаются на полгода позже версий Delphi с аналогичными номерами. Поэтому каждая версия C++Builder совершеннее аналогичной версии Delphi, но слабее последующей версии Delphi.

Впрочем, основное различие Delphi и C++Builder не в этом, а в языках программирования, которые лежат в их основе. Delphi базируется на языке Object Pascal, а C++Builder — на языке C++. Эти языки, сначала существенно различные по своим возможностям, со временем все более сближаются. Сейчас оба они представляют прекрасные инструменты объектно-ориентированного программирования, различающиеся, в основном, синтаксисом. Но C++ все-таки богаче и опережает аналогичные версии Object Pascal. С этой точки зрения он предпочтительнее. Правда, эти преимущества C++ перед Object Pascal проявляются только в достаточно сложных приложениях.

Фирма Borland позаботилась о том, чтобы приложения, разработанные на C++Builder и на Delphi можно было достаточно просто конвертировать друг в друга. Таким образом, в одной из этих систем вы можете использовать свои наработки, сделанные в другой системе.

Отдельно надо сказать об одной из главных задач C++Builder и Delphi — разработке приложений для работы с базами данных. В этой области C++Builder и Delphi занимают самые передовые позиции, работая с любыми системами управления базами данных.

В целом C++Builder — великолепный инструмент как для начинающих программистов, так и для асов программирования. Так что не зря вы заинтересовались C++Builder и, надеюсь, с пользой ознакомитесь с данной книгой.

1.5 Язык объектно-ориентированного проектирования C++

1.5.1 Введение

Возможности объектно-ориентированного проектирования в C++Builder базируются на свойствах языка C++. Детальное изучение C++ выходит за рамки данной книги. Впрочем, все необходимые вам для чтения этой книги сведения о C++ вы можете почерпнуть в справочной части книги в главах 12, 13, 15. Кроме того, отдельные аспекты языка не раз будут обсуждаться в различных главах книги. В целом вы получите достаточно сведений для написания большинства приложений, с которыми вас столкнет жизнь. Ну а желающим глубоко изучить C++ и научиться профессионально использовать все его возможности придется изучить его по специальной литературе.

Язык C++ — это дальнейшее развитие более раннего языка программирования C. Если вы совсем не знакомы ни с одной версией семейства языков C, то можно рекомендовать вам прямо сейчас, прежде, чем читать эту книгу далее, посмотреть в главе 12 раздел 12.1, в котором описан синтаксис этого языка, и бегло просмотреть разделы, в которых описано объявление констант, переменных и функций. Впредь обращайтесь к главам 12 и 13, когда что-то в изложении того или иного вопроса покажется вам неясным. Я надеюсь, что это будет происходить редко и главы 13 и 14 пригодятся вам, в основном, только для углубленного изучения каких-то тем.

В данной главе мы ограничим знакомство с C++ только вопросами общей организации программы и доступа к объектам, их свойствам и методам.

1.5.2 Общие сведения о программах на C++

Программа на C++ состоит из *объявлений* (переменных, констант, типов, классов, функций) и *описаний функций*. Среди функций всегда имеется главная — **main** для консольных приложений (работающих с WIN32) или **WinMain** для приложений Windows. Именно эта главная функция выполняется после начала работы программы. Обычно в C++Builder эта функция очень короткая и выполняет только некоторые подготовительные операции, необходимые для начала работы. А далее при объектно-ориентированном подходе работа приложения определяется происходящими событиями и реакцией на них объектов.

Как правило, программы строятся по модульному принципу и состоят из множества *модулей*. Принцип модульности очень важен для создания надежных и относительно легко модифицируемых и сопровождаемых приложений. Четкое соблюдение принципов модульности в сочетании с принципом скрытия информации позволяет внутри любого модуля проводить какие-то модификации, не затрагивая при этом остальных модулей и головную программу.

Все объекты компонентов размещаются в объектах — формах. Для каждой формы, которую вы проектируете в своем приложении, C++Builder создает отдельный модуль. Именно в модулях и осуществляется программирование задачи. В обработчиках событий объектов — форм и компонентов, вы помещаете все свои алгоритмы. В основном они сводятся к обработке информации, содержащейся в свойствах одних объектов, и задании по результатам обработки свойств других объектов. При этом вы постоянно обращаетесь к методам различных объектов. Вопросами доступа к свойствам и методам объектов мы и займемся в дальнейших разделах данной главы.

Согласно принципам скрытия информации обычно текст модуля разделяют на *заголовочный файл* интерфейса, который содержит объявления классов, функ-

ций, переменных и т.п., и *файл реализации*, в котором содержится описание функций. Стандартное расширение файлов реализации — **.cpp**. Стандартное расширение заголовочных файлов — **.h**.

После того, как программа написана, на ее основе должен быть создан *выполняемый файл* (модуль). Этот процесс осуществляется в несколько этапов.

Сначала работает *препроцессор*, который преобразует исходный текст. Препроцессор осуществляет преобразования в соответствии со специальными *директивами препроцессора*, которые размещаются в исходном тексте. Препроцессор может в соответствии с этими директивами включать тексты одних файлов в тексты других, разворачивать *макросы* — сокращенные обозначения различных выражений и выполнять множество других преобразований.

После окончания работы препроцессора начинает работать *компилятор*. Его задача — перевести тексты модулей в машинный (объектный) код. В результате для каждого исходного файла **.cpp** создается объектный файл, имеющий расширение **.obj**.

После окончания работы компилятора работает *компоновщик*, который объединяет объектные файлы в единый загрузочный выполняемый модуль, имеющий расширение **.exe**. Этот модуль можно запускать на выполнение.

1.5.3 Структура головного файла проекта

В процессе проектирования вами приложения C++Builder автоматически создает коды головного файла проекта, коды отдельных модулей и коды их заголовочных файлов. Головной файл проекта содержит функцию **WinMain** (в дальнейшем в этой книге будут рассматриваться практически только приложения для Windows, использующие именно эту функцию, а не функцию **main**). В прочие модули вы вводите свой код, создавая обработчики различных событий. В заголовочные файлы этих модулей вы вводите свои объявления. Но головной модуль, как правило, вы не трогаете и даже не видите его текст. Только в исключительных случаях вам надо что-то изменить в тексте головного модуля, сгенерированном C++Builder. Тем не менее, хотя бы ради этих исключительных случаев, надо все-таки представлять вид головного файла проекта и понимать, что означают его операторы.

Чтобы увидеть код головного файла проекта, надо выполнить в среде разработки C++Builder команду Project | View Source. Типичный головной файл проекта имеет следующий вид (в приведенный текст добавлены русские комментарии):

```
//
// директивы препроцессора
#include <vcl.h>
#pragma hdrstop

// макросы, подключающие файлы ресурсов и форм
USERES("Project1.res");
USEFORM("Unit1.cpp", Form1);
USEFORM("Unit2.cpp", Form2);
//

// функция main
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TForm1), &Form1);
        Application->CreateForm(__classid(TForm2), &Form2);
        Application->Run();
    }
}
```

```
}  
catch (Exception &exception)  
{  
    Application->ShowException(&exception);  
}  
return 0;  
}
```

Начинается файл головного модуля строками, первый символ которых — '#'. С этого символа начинаются директивы препроцессора (см. подробности в разделе 12.2 главы 12). Среди них наиболее важны для вас директивы **#include**, с которыми вы еще не раз будете иметь дело. Эти директивы подключают в данный файл тексты указанных в них файлов. В частности, подобными директивами включаются в текст заголовочные файлы. Например, директива **#include <vcl.h>** подключает заголовочный файл **vcl.h**, содержащий объявления, используемые в библиотеке визуальных компонентов C++Builder.

После директив препроцессора в файле размещены предложения **USERES** и **USEFORM**. Это макросы, используемые для подключения к проекту файлов форм, ресурсов и др. Препроцессор развернет эти макросы в соответствующий код. В данном случае вы можете видеть два макроса **USEFORM**, подключающих формы. C++Builder автоматически формирует соответствующее предложение с макросом **USEFORM** для каждой формы, вносимой вами в проект. Первый параметр макроса содержит имя файла модуля, соответствующего форме (например, «Unit1.cpp»), а второй параметр — имя формы.

После всех этих вспомогательных предложений в файле расположена главная функция программы — **WinMain**. Ее первым параметром является дескриптор данного экземпляра приложения. Дескриптор — это некий уникальный указатель, позволяющий Windows разбираться в множестве одновременно открытых окон различных приложений. Иногда мы будем использовать дескрипторы при обращении к различным функциям API Windows (API Windows — это пользовательский интерфейс Windows, содержащий множество полезных функций). Второй параметр **WinMain** — дескриптор предыдущего экземпляра вашего приложения (если пользователь выполняет одновременно несколько таких приложений). Третий параметр является указателем на строку с нулевым символом в конце, содержащую параметры, передаваемые в программу через командную строку. Иногда такие параметры используются для переключения режимов работы программы или для задания различных опций при запуске приложения из диспетчера программ или функций **WinExec**. Последний параметр определяет окно приложения. Этот параметр может в дальнейшем передаваться в функцию **ShowWindow**.

Все эти параметры функции **WinMain** используются достаточно редко, так что пока можете не стараться разбираться в их назначении.

После заголовка функции **WinMain** следует ее тело, заключенное в фигурные скобки. Первый выполняемый оператор тела функции — **Application->Initialize** инициализирует объекты компонентов данного приложения. Последующие операторы **Application->CreateForm** создают объекты соответствующих форм. Формы создаются в той последовательности, в которой следуют эти операторы. Первая из создаваемых форм является главной.

Последний оператор — **Application->Run** начинает собственно выполнение программы. После этого оператора программа ждет соответствующих событий, которые и управляют ее ходом.

Перечисленные операторы тела функции **WinMain** заключены в блок **try**, после которого следует блок **catch**. Это структура, связанная с обработкой так называемых исключений — аварийных ситуаций, возникающих при работе программы. Если такая аварийная ситуация возникнет, то будут выполнены операторы, расположенные в блоке **catch**. По умолчанию в этом блоке расположен стандартный обработчик исключений с помощью функции **Application->ShowException**.

Подробнее об обработке исключений вы можете посмотреть в главе 12 в разделе 12.10.

Последним оператором тела функции **WinMain** является оператор **return(0)**, завершающий приложение с кодом завершения 0.

Все описанные выше операторы головного файла приложения заносятся в него автоматически в процессе проектирования вами приложения. Например, при добавлении в проект новой формы в файл автоматически вставляются соответствующее предложение **USEFORM** и оператор **Application->CreateForm**, создающий форму. Так что обычно ничего в головном файле изменять не надо и даже нет необходимости его смотреть.

Если вам надо ввести какой-то свой текст в головной модуль, вы можете сделать это, введя объявления необходимых констант, переменных, функций, добавить или изменить операторы в теле функции. Например, вам может потребоваться при запуске приложения на выполнение провести какие-то настройки (например, настроить формы на тот или иной язык — русский или английский). Или сделать какой-то запрос пользователю и в зависимости от ответа создавать или не создавать те или иные формы. Или проанализировать параметры, переданные в программу через командную строку.

Посмотрим, как можно вводить собственный код в головной файл. Те, кому изложенное ниже будет в данный момент мало понятно, могут пропустить следующие несколько абзацев и вернуться к ним позднее, когда лучше освоятся с программированием на языке C++, а главное — когда возникнет достаточно экзотичная необходимость изменять что-то в головном файле.

Пусть, например, вы хотите, чтобы вторая форма вашего приложения **Form2** создавалась только в случае, если при запуске приложения через командную строку в него передана опция **Y**. В этом случае вы можете изменить заголовок функции **WinMain** следующим образом:

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR S, int)
```

Этот заголовок объявляет строку **S**, в которой будет содержаться текст командной строки. Тогда приведенный выше оператор

```
Application->CreateForm(__classid(TForm2), &Form2);
```

можно заменить оператором

```
if (S[0] == 'Y')
    Application->CreateForm(__classid(TForm2), &Form2);
```

В этом случае, если ваше приложение **Project1** будет запускаться командой **Project1 Y**, то форма **Form2** будет создаваться. В остальных случаях этой формы не будет.

Аналогичный выбор можно сделать, не анализируя командную строку, а предложив пользователю соответствующий вопрос. В этом случае приведенный выше оператор можно заменить следующим (о методе **MessageBox** см. в главе 15 в разделе 15.7.2.3):

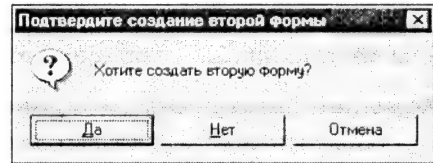
```
if (Application->MessageBox(
    "Хотите создать вторую форму?",
    "Подтвердите создание второй формы",
    MB_YESNOCANCEL + MB_ICONQUESTION) == IDYES)
    Application->CreateForm(__classid(TForm2), &Form2);
```

Тогда в момент запуска приложения пользователю будет сделан запрос, показанный на рис. 1.3. При положительном ответе пользователя форма будет создана, в противном случае — нет.

Вы можете, конечно, ввести в головной файл программы и другие операторы, функции и т.п. Все это можно сделать, но это будет плохой стиль программирования, поскольку он противоречит принципу модульности. Выше уже говорилось о

Рис. 1.3

Окно запроса пользователю



важности соблюдения этого принципа. Все необходимые вам в начале выполнения процедуры и функции настройки помещайте в один из модулей форм, а еще лучше — в отдельный модуль без формы. Такой модуль, не связанный с какой-то формой, можно включить в приложение, выполнив в среде разработки C++Builder команду **File | New** и щелкнув на пиктограмме **Unit**. В этом или ином модуле вы можете предусмотреть функцию, которая осуществляет все необходимые настройки. Тогда в головной программе достаточно будет вызвать в соответствующий момент эту функцию, передав в нее, если необходимо, какие-то параметры, например, текст командной строки.

Хороший стиль программирования

Не перегружайте головной файл проекта никаким дополнительным кодом. Все необходимые настройки, которые должны осуществляться в начале выполнения приложения, помещайте в отдельный модуль. В этом случае в головном модуле достаточно поместить вызов предусмотренной вами функции настройки.

Предупреждение

Учтите, что все определенные вами в головном файле проекта глобальные константы и переменные будут доступны в другом блоке только в случае, если они объявлены там со спецификацией **extern** (см. главу 12 раздел 12.4.2). Функции, определенные вами в головном файле проекта, будут доступны в другом блоке только в случае, если там повторен их прототип (см. главу 12 раздел 12.5.1).

Пусть, например, вы написали некоторую функцию, назвав ее **begin**, в которой проводится настройка программы в зависимости от опций, переданных через командную строку. И пусть вы поместили эту функцию в модуль **Unit1.cpp**, а ее объявление — в заголовочный файл этого модуля **Unit1.h** (вне описания класса — см. раздел 1.5.4). Тогда вы можете в головной файл приложения включить директиву препроцессора

```
#include "Unit1.h"
```

подключающую файл **Unit1.h**, изменить заголовок функции **WinMain** на

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR S, int)
```

т.е. ввести параметр **S**, воспринимающий командную строку, и поставить первым выполняемым оператором функции **WinMain** оператор:

```
begin(S);
```

вызывающий функцию **begin** и передающий в нее текст командной строки.

Сама функция **begin** может иметь вид:

```
void begin(String s)
{
    // операторы анализа командной строки и настройки
}
```

Таким образом вы, не нарушив принципа модульности, можете осуществить все действия, необходимые вам в начале работы программы.

Имя головного файла проекта по умолчанию дается стандартное: **Project1**, **Project2** и т.п. Это же имя будет и у выполняемого модуля вашей программы. Так

что желательно изменить имя по умолчанию. Для этого достаточно сохранить головной файл проекта под соответствующим именем.

Хороший стиль программирования

Всегда сохраняйте проект под каким-то осмысленным именем, изменяя тем самым имя проекта, заданное C++Builder по умолчанию. Иначе очень скоро вы запутаетесь в бесконечных программах Project1, лежащих в различных ваших каталогах.

1.5.4 Структура файлов модулей форм

Рассмотрим теперь, как выглядят тексты модулей форм. Каждый такой модуль состоит из двух файлов: заголовочного, содержащего описание класса формы, и файла реализации. Ниже приведены тексты этих файлов модуля формы, на которой размещена одна метка (компонент типа **TLabel**) и одна кнопка (компонент типа **TButton**). Подробные комментарии в этом тексте поясняют, куда и что в этот код вы можете добавлять.

Заголовочный файл:

```
//-----
#ifndef Unit1H
#define Unit1H
//-----
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
// сюда могут помещаться дополнительные директивы
// препроцессора (в частности, include),
// не включаемые в файл автоматически

//-----
// объявление класса формы TForm1
class TForm1 : public TForm
{
    __published:        // IDE-managed Components
    // размещенные на форме компоненты
    TButton *Button1;
    TLabel *Label1;
    void __fastcall Button1Click(TObject *Sender);

private: // User declarations
    // закрытый раздел класса
    // сюда могут помещаться объявления типов, переменных, функций,
    // включаемых в класс формы, но не доступных для других модулей

public:        // User declarations
    // открытый раздел класса
    // сюда могут помещаться объявления типов, переменных, функций,
    // включаемых в класс формы и доступных для других модулей
    __fastcall TForm1(TComponent* Owner);
};
//-----
extern PACKAGE TForm1 *Form1;
//-----
// сюда могут помещаться объявления типов, переменных, функций,
// которые не включаются в класс формы;
// доступ к ним из других блоков возможен только при соблюдении
// некоторых дополнительных условий
#endif
```

Файл реализации:

```
//-----
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"

// сюда могут помещаться дополнительные директивы
// препроцессора (в частности, include),
// не включаемые в файл автоматически

// объявление объекта формы Form1
TForm1 *Form1;
//-----
// вызов конструктора формы Form1
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    // сюда вы можете поместить операторы,
    // которые должны выполняться при создании формы
}
//-----
// сюда могут помещаться объявления типов и переменных,
// доступ к которым из других модулей возможен только при
// соблюдении некоторых дополнительных условий;
// тут же должны быть реализации всех функций, объявленных в
// заголовочном файле, а также могут быть реализации любых
// дополнительных функций, не объявленных ранее

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Close();
}
```

Рассмотрим подробнее эти файлы. Заголовочный файл начинается с автоматически включенных в него директив препроцессора. В частности, C++Builder сам помещает тут директивы **include**, подключающие копии файлов, в которых описаны те компоненты, переменные, константы, функции, которые вы используете в данном модуле. Однако, для некоторых функций такое автоматическое подключение не производится. В этих случаях разработчик должен добавить соответствующие директивы **include** вручную.

После директив препроцессора следует описание класса формы. Имя класса вашей формы — **TForm1**. Класс содержит три раздела: **__published** — открытый раздел, содержащий объявления размещенных на форме компонентов и обработчиков событий в них, **private** — закрытый раздел класса, и **public** — открытый раздел класса. В данном случае в разделе **__published** вы можете видеть объявления указателей на два компонента: компонент **Button1** типа **TButton** и компонент **Label1** типа **TLabel**. Там же вы видите объявление функции **Button1Click** — введенного пользователем обработчика события щелчка на кнопке **Button1**. Все, что имеется в разделе **__published**, C++Builder включает в него автоматически в процессе проектирования вами формы. Так что вам не приходится, как правило, работать с этим разделом. А в разделы **private** и **public** вы можете добавлять свои объявления типов, переменных, функций. То, что вы или C++Builder объявите в разделе **public**, будет доступно для других классов и модулей. То, что объявлено в разделе **private**, доступно только в пределах данного модуля. Как вы можете видеть, единственное, что C++Builder самостоятельно включил в раздел **public**, это объявление (прототип) конструктора вашей формы **TForm1**.

После объявления класса следует предложение **PACKAGE**, которое включает-ся в файл автоматически и которое мы сейчас рассматривать не будем. После этого вы можете разместить объявления типов, переменных, функций, к которым при соблюдении некоторых дополнительных условий (см. раздел 1.5.5) будет доступ из других модулей, но которые не включаются в класс формы.

Теперь рассмотрим текст файла реализации модуля. После автоматически включенных в этот файл директив препроцессора следует тоже автоматически включенное объявление указателя на объект формы **Form1**, а затем — вызов конструктора формы. Тело соответствующей функции пустое, но вы можете включить в него какие-то операторы. После этого размещаются описания всех функций, объявленных в заголовочном файле. Вы можете также размещать здесь объявления любых типов, констант, переменных, не объявленных в заголовочном файле, и размещать описания любых функций, не упомянутых в заголовочном файле.

Имена файлам модулей C++Builder дает по умолчанию: для первого модуля имя равно **Unit1**, для второго **Unit2** — и т.д.

Хороший стиль программирования

Если в вашем приложении несколько модулей, сохраняйте их файлы под какими-то осмысленными именами, изменяя тем самым имена модулей, заданные C++Builder по умолчанию. Вам проще будет работать с осмысленными именами, а не с именами Unit1, Unit2, Unit3, которые ни о чем не говорят.

1.5.5 Области видимости и доступ к объектам, переменным и функциям модуля

1.5.5.1 Пример модуля, содержащего объекты и процедуры

Теперь посмотрим, как можно вводить в модуль переменные, функции и осуществлять к ним доступ. Ниже приведен текст кода модуля, в котором на форме размещены два компонента: кнопка **Button1** типа **TButton** и метка **Label1** типа **TLabel**. Кроме того, в модуле введен обработчик события, связанного со щелчком пользователя на кнопке, и в разных местах модуля введены переменные и функции, чтобы можно было видеть, как получить к ним доступ.

Заголовочный файл:

```
//-----
#ifndef Unit1H
#define Unit1H
//-----
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//-----
class TForm1 : public TForm
{
__published:      // IDE-managed Components
    TLabel *Label1;
    TButton *Button1;
    void __fastcall Button1Click(TObject *Sender);

private:          // User declarations
    //Функция F1 и переменная Ch6 доступны только в данном модуле
    void __fastcall F1(char Ch);
    char Ch6;

public:            // User declarations
```

```

    __fastcall TForm1(TComponent* Owner);
// Переменная Ch1 и функция F2 доступны для объектов
// любых классов и для других модулей, но со ссылкой
// на объект
    char Ch1;
    void __fastcall F2(char Ch);
};
//-----
extern PACKAGE TForm1 *Form1;

// Глобальная переменная Ch2 и функция F3 доступны в пределах
// данного модуля; переменная Ch2 доступна в других модулях,
// если определена там со спецификацией extern;
// функция F3 доступна в других модулях, если там содержится
// ее прототип
    char Ch2;
    void F3(char Ch);
#endif

Файл реализации:

//-----
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//-----
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//-----
// Глобальная переменная Ch3 и функция F4 доступны в пределах
// данного модуля; переменная Ch3 доступна в других модулях,
// если определена там со спецификацией extern;
// функция F4 доступна в других модулях, если там содержится
// ее прототип
char Ch3;

void F4(char Ch)
{
    Form1->Label1->Caption = Form1->Label1->Caption + Ch +
        Form1->Ch1;
}
//-----
void __fastcall TForm1::F1(char Ch)
{
    Label1->Caption = Label1->Caption + Ch + Ch1;
}
//-----
void __fastcall TForm1::F2(char Ch)
{
    Label1->Caption = Label1->Caption + Ch + Ch1;
}
//-----
void F3(char Ch)
{

```

```

Form1->Label1->Caption = Form1->Label1->Caption + Ch +
                        Form1->Ch1;
}
//_____
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    // Переменная Ch4 доступна только внутри данной функции
    char Ch4;
    Ch1 = '-';
    Ch2 = 'A';
    Ch3 = 'B';
    Ch4 = 'C';
    Label1->Caption = "";
    F1(Ch1);
    F2(Ch2);
    F3(Ch3);
    F4(Ch4);
    Label1->Font->Color = clRed;
}

```

Помимо функции **Button1Click** обработки щелчка на кнопке **Button1** в код занесено в разные места модуля еще несколько одинаковых функций: **F1** — **F4**, и несколько переменных символьного типа: **Ch1** — **Ch4**. Сейчас мы не будем разбираться подробно в самих функциях. Нам важно понять, как из функций обращаться к различным объектам и переменным. Но краткое описание того, что делают эти функции, надо дать.

У компонента — метки типа **TLabel** имеется свойство **Caption** — надпись на метке. Каждая из функций **F1** — **F4** берет значение этой надписи, прибавляет к ней символ, переданный в нее как параметр **Ch**, прибавляет далее символ, хранящийся в переменной **Ch1**, и возвращает надпись с этими добавлениями обратно в метку.

Обработчик щелчка на кнопке — функция **TForm1::Button1Click**, задает символьным переменным **Ch1** — **Ch4** значения символов «-», «A», «B» и «C», затем очищает свойство **Caption** метки **Label1**, занося в него пустую строку, а затем поочередно обращается к функциям **F1** — **F4**, передавая в них в качестве параметров различные символы. В заключение надпись метки окрашивается в красный цвет. Для этого используется свойство **Font** — шрифт объекта **Label1**. Это свойство само является объектом, имеющим свойство **Color** — цвет. Значение этого свойства изменяет последний оператор процедуры **TForm1::Button1Click**.

1.5.5.2 Доступ к свойствам и методам объектов

Рассмотрим теперь, как получить из программы доступ к свойствам и методам объектов. Доступ к интересующим нас объектам — компонентам можно получить через объявленные в заголовочном файле модуля указатели на эти объекты. Например, в объявлении класса формы **TForm1** в заголовочном файле имеется строка

```
TLabel *Label1;
```

Эта строка объявляет **Label1** как указатель на метку — объект типа **TLabel**. Если вы плохо представляете, что такое указатели, посмотрите в главе 13 раздел 13.7, или просто отнеситесь к тому, о чем будет рассказано ниже, как к некоторому обязательному формализму. Честно говоря, программы в C++Builder часто можно писать, не задумываясь о сути этого формализма.

Доступ к элементам класса (данным — свойствам и функциям — методам) обеспечивается одним из следующих двух способов. Можно использовать *операцию стрелка* (символ «-» и символ «>», записанные без пробела, т.е. «->») или *операцию точку* (.). Первая из них применяется при обращении к объекту через указатель на него, вторая — при обращении по имени переменной объекта или по ссылке на него.

Посмотрите, например, в приведенном выше файле реализации модуля тексты функций **F1** и **F2**. Вы увидите в них выражения вида **Label1->Caption**. Они означают: свойство **Caption** объекта **Label1**. То же самое свойство **Caption** можно было бы получить и с помощью выражения **(*Label1).Caption**. Здесь операция разыменованье указателя **(*Label1)** дает сам объект, к которому можно применять операцию точка.

Хотя эти два способа доступа к свойствам и методам объекта эквивалентны, обычно в C++Builder используется операция стрелка. Поэтому в дальнейшем изложении в данной книге мы практически всегда будем пользоваться этой формой записи. И если вы не очень представляете себе, что такое указатели, и не хотите пока смотреть пояснения в главе 13 в разделе 13.7, то просто запомните, что ко всем свойствам и методам объектов надо обращаться, включая перед ними символы **<->**.

Иногда свойство объекта является в свою очередь объектом. Тогда в обращении к этому свойству указывается вся цепочка предшествующих объектов. Например, метки имеют свойство **Font** — шрифт, которое в свою очередь является объектом. У этого объекта имеется множество свойств, в частности, свойство **Color** — цвет шрифта. Чтобы сослаться на цвет шрифта метки **Label1**, надо написать **Label1->Font->Color** (см. в тексте примера функцию **TForm1::Button1Click**). Это означает: свойство **Color** объекта **Font**, принадлежащего объекту **Label1**.

Аналогичная нотация используется и для доступа к методам объекта. Например, для метки, как и для большинства других объектов, определен метод **Hide**, который делает метку невидимой. Если вы в какой-то момент решили сделать метку **Label1** невидимой, можете написать оператор

```
Label1->Hide();
```

1.5.5.3 Различие переменных и функций, включенных и не включенных в описание класса

Теперь посмотрим, чем различаются переменные и функции, включенные и не включенные в описание класса. Переменные и функции, включенные в описание класса, обычно называются соответственно *данными-элементами* и *функциями-элементами*. Применительно к объектно-ориентированному проектированию в C++Builder чаще их называют свойствами и методами. В приведенном в разделе 1.5.5.1 примере переменная **Ch1** и функции **F1** и **F2** включены в описание класса, а переменные **Ch2**, **Ch3** и функции **F3** и **F4** объявлены вне класса. В чем будет проявляться различие в их использовании?

Если в приложении создается только один объект данного класса (в нашем примере — только один объект формы класса **TForm1**), то различие в основном чисто внешнее. Для функций, объявленных в классе, в их описании к имени функции должна добавляться ссылка на класс с помощью так называемой бинарной операции разрешения области действия (**::**). В нашем примере имена функций **F1**, **F2** и **Button1Click** в описании этих функций заменяются на **TForm1::F1**, **TForm1::F2** и **TForm1::Button1Click**. Тем самым указывается, что речь идет о функциях класса **TForm1**. Для функций **F3** и **F4**, объявленных вне класса, такого дополнения к имени не требуется.

Необходимость добавления в имена функций, описанных в классе, ссылок на класс объясняется просто. Вы можете вне класса описать другую свою функцию с тем же именем (например, **F1**), что и у функции класса. И тогда из функций, не описанных в классе, вы сможете сослаться на обе эти функции **F1**, только на одну из них непосредственно — по имени **F1**, а на другую через объект класса — **Form1::F1**. Благодаря этому, при описании своих функций вне класса вы можете даже не знать имен всех функций, описанных в классе (может быть этот класс опи-

сан в другом модуле, текст которого вы не видели). Никакой путаницы при этом не возникнет.

Таким образом, применение операции разрешения области действия позволяет объявить в разных классах и вне классов переменные и функции с одинаковыми именами. В этом случае операция разрешения области действия указывает, о какой именно переменной или функции идет речь. В некоторых случаях при работе с C++Builder разрешение области действия приходится делать вручную. Это происходит в тех случаях, когда компилятор выдает сообщение, что не может выбрать одну из нескольких альтернатив, например, не знает, к какому классу относится указанный вами метод.

Обращение к переменным и функциям, описанным внутри и вне класса, из функций, описанных вне класса, различается. К переменным и функциям, описанным вне класса, обращение происходит просто по их именам, а к переменным и функциям, описанным в классе, через имя объекта класса. Поэтому в нашем примере в функциях **F3** и **F4** обращение к переменной **Ch1** имеет вид **Form1->Ch1**. По той же причине и обращение к свойству **Caption** объекта **Label1** в этих функциях имеет вид **Form1->Label1->Caption**. Только через ссылку на объект **Form1** внешние по отношению к классу функции могут получить доступ ко всему, объявленному в классе.

Все эти ссылки на объект не требуются в функциях, объявленных в классе. Поэтому в функциях **TForm1::F1**, **TForm1::F2** и **TForm1::Button1Click** ссылки на переменную **Ch1** и на объект **Label1** не содержат дополнительных ссылок на объект формы.

Если в приложении создается несколько объектов одного класса, например, несколько форм класса **TForm1** (как вы увидите впоследствии, это часто делается в приложениях с интерфейсом множества документов MDI), то проявляются более принципиальные различия между переменными, описанными внутри и вне класса. Переменные вне класса (в нашем примере **Ch2** и **Ch3**) так и остаются в одном экземпляре. А переменные, описанные в классе (в нашем примере **Ch1**), тиражируются столько раз, сколько объектов данного класса создано. Т.е. в каждом объекте класса **TForm1** будет своя переменная **Ch1** и все они друг с другом никак не будут связаны. Таким образом, в переменную, описанную внутри класса, можно заносить какую-то информацию, индивидуальную для каждого объекта данного класса. А переменная, описанная в модуле вне описания класса, может хранить только одно значение.

Отметим без деталей пояснений еще одну особенность, которую вы уже, вероятно, заметили в приведенных текстах файлов. Перед именами функций-элементов класса ставится опция компилятора **__fastcall**. Не вдаваясь в детали скажем, что эта опция влияет на процесс компиляции и обеспечивает передачу параметров функции в быстрые регистры, что ускоряет вызов функции. Для функций-элементов классов эту опцию следует указывать всегда. Для других функций ее можно указывать, а можно и не указывать. Впрочем, эту опцию целесообразно указывать и для функций, не являющихся элементами класса (в приведенном примере это не сделано просто чтобы подчеркнуть различие между функциями-элементами и прочими функциями).

В заключение просуммируем изложенные правила.

- В описание функций-элементов должна добавляться ссылка на класс с помощью операции разрешения области действия (::).
- Функции-элементы объявляются и описываются с применением опции компилятора **__fastcall**.
- В функциях-элементах обращение к другим функциям-элементам и данным-элементам того же класса может осуществляться без указания на объект.

- В функциях, не являющихся элементами класса данного объекта, доступ к функциям-элементам (методам) и данным-элементам (свойствам) осуществляется через указатель на объект с помощью операции стрелка (->) или (значительно реже) с помощью разыменования указателя на объект и операции точка (.).

1.5.5.4 Области видимости переменных и функций

Теперь остановимся на вопросе об областях видимости или областях действия элементов программы — переменных и функций, т.е. о связи места их объявления в программе и места их использования. Частично этот вопрос мы уже затрагивали в предыдущем разделе, не упоминая о самом понятии *область видимости*.

При решении вопросов видимости важнейшее значение имеет понятие блока. Блок — это фрагмент кода, ограниченный фигурными скобками «{ }». Переменные, объявленные вне какого-то блока, являются глобальными и имеют область видимости файл. Если какая-то переменная объявлена внутри блока, то ее область видимости — блок. Т.е. это локальная переменная, которую можно использовать только от момента ее объявления до первой встретившейся в коде закрывающей фигурной скобки «}». Если имеются вложенные блоки, то переменная видна и в этих вложенных блоках. Если в каком-то блоке объявлена переменная с тем же именем, какое имеет глобальная переменная или переменная, объявленная во внешнем блоке, то это внутреннее объявление делает невидимой одноименную внешнюю переменную.

Например:

```
int i = 1, k = 4    // объявление глобальных переменных
{
    int i = 5, j = 2; // объявление переменных внешнего блока
    ...              // видны переменные j, k
                      // и переменная i этого блока
    {
        int i = 7;    // объявление переменной i внутреннего блока
        ...           // видны переменные j, k
                      // и переменная i внутренняя
    }
    ...              // видны переменные j, k
                      // и переменная i внешнего блока
}
```

Локальная переменная не только видима в пределах блока, в котором она объявлена. Ее время жизни тоже определяется временем выполнения блока. Переменная создается в момент входа в блок и разрушается в тот момент, когда управление выходит за пределы блока. Таким образом подобная переменная не может сохранять какие-то значения в промежутках между выполнением операторов блока. В приведенном выше примере переменная *i* во внутреннем блоке будет создаваться каждый раз, когда управление передается в этот блок, ей каждый раз будет присваиваться значение 7 и она каждый раз будет разрушаться при выходе из блока.

Сказанное выше о времени жизни относится к так называемым автоматическим (**auto**) переменным и не относится к статическим переменным, объявленным как **static**. Например:

```
static int i = 7;
```

Такие статические переменные существуют все время работы программы и инициализируются только один раз. Таким образом, в этих переменных можно накапливать какую-то информацию. Например, они могут служить счетчиками числа обращений к блоку.

Выше говорилось, что если во внутреннем блоке объявлена переменная с тем же именем, что во внешнем блоке, или с тем же именем, что и глобальная переменная, то соответствующая внешняя или глобальная переменная в блоке не видна. Однако, в C++ имеется унарная операция разрешения области действия, позволяющая все-таки во внутреннем блоке обращаться к глобальной переменной с тем же идентификатором. Операция обозначается двумя символами двоеточия «::». Если поставить эти символы перед идентификатором переменной, то обращение будет к глобальной переменной с этим именем. Так в приведенном выше примере во внутреннем блоке можно, например записать оператор

```
i = ::i + 1;
```

Этот оператор присвоит внутренней переменной `i` значение на единицу большее значения глобальной переменной `i`.

Подчеркнем, что таким образом можно получить доступ только к одноименной глобальной переменной, а не к локальной переменной, описанной во внешнем блоке.

Теперь остановимся на проблемах видимости переменных в приложениях, имеющих несколько модулей. Пусть вы имеете два модуля — **Unit1** и **Unit2** и хотите в модуле **Unit2** видеть и использовать переменные и функции, объявленные в модуле **Unit1**. Вы можете в модуле **Unit2** видеть те переменные, которые являются глобальными в модуле **Unit1**, т.е. объявлены вне каких-нибудь функций в заголовочном файле модуля или в его файле реализации. Но для того, чтобы это было возможно, вы должны повторно объявить их (без инициализации) в модуле **Unit2** со спецификацией **extern**. Например, если в модуле **Unit1** имеется объявление глобальной переменной

```
int a1 = 10;
```

то в модуле **Unit2** вы можете использовать эту переменную, если запишете объявление

```
extern int a1;
```

Причем, это не зависит от того, включили ли вы директивой **#include** заголовочный файл **Unit1.h** в модуль **Unit2**, или нет.

Отметим еще одну особенность использования переменных, описанных в другом модуле. Если в заголовочном модуле **Unit1** объявлена описанная выше переменная **a1**, а в модуле **Unit2** вы включили директивой **#include** заголовочный файл **Unit1.h**, но не записали объявление этой переменной со спецификацией **extern** (вообще не дали объявление **a1**), то в модуле **Unit2** будет создана копия переменной **a1**, инициализированная согласно объявлению в **Unit1**. Но это будет копия, совершенно изолированная от переменной **a1** в модуле **Unit1**. В модулях **Unit1** и **Unit2** будут существовать две различные переменные с одним именем **a1**. И изменение одной из них никак не скажется на значении другой.

Все сказанное относится только к глобальным переменным. Локальные переменные, объявляемые внутри функций, невозможно видеть в другом модуле.

Теперь рассмотрим видимость функций в приложениях, имеющих несколько модулей. Если в модуле **Unit1** в его заголовочном файле вне описания класса вы объявили некоторую функцию **F**, то в другом модуле **Unit2** вы можете использовать ее при выполнении одного из двух условий:

- вы включаете директивой **#include** в модуль **Unit2** заголовочный файл **Unit1.h**
- вы повторяете в модуле **Unit2** (в заголовочном файле или файле реализации) объявление функции **F**

В обоих случаях вы сможете вызвать функцию **F** из любого места модуля **Unit2**.

Если же функция **F** объявлена в модуле **Unit1** не заголовочном файле, а в файле реализации, то единственный способ использовать ее в модуле **Unit2** — повторить в нем объявление функции.

Если вы хотите предотвратить возможность обращения к функции из другого модуля, ее надо объявить со спецификацией **static**. Например:

```
static void F(void);
```

Подведем некоторые итоги проведенного рассмотрения проблем видимости переменных и функций.

- Переменные, объявленные в заголовочном файле модуля или в файле его реализации вне описания класса и функций, являются глобальными. Они доступны везде внутри данного модуля. Для доступа к ним из внешних модулей в этих модулях должно быть повторено их объявление (без инициализации) с добавлением спецификации **extern**. В рассмотренном примере это относится к переменным **Ch2** и **Ch3**.
- Функции, объявленные в заголовочном файле модуля вне описания класса, являются глобальными. Они доступны везде внутри данного модуля. Для доступа к ним из внешних модулей в этих модулях или надо повторить их объявление, или включить директивой **#include** заголовочный файл того модуля, в котором функции описаны. В рассмотренном примере это относится к функции **F3**.
- Функции, объявленные в файле реализации модуля, являются глобальными. Они доступны везде внутри данного модуля. Для доступа к ним из внешних модулей в этих модулях надо повторить их объявление. В рассмотренном примере это относится к функции **F4**.
- Элементы (переменные и функции), объявленные в классе в разделе **private**, видимы и доступны только внутри данного модуля. При этом из функций, объявленных внутри класса, к ним можно обращаться непосредственно по имени, а из других функций — только со ссылкой на объект данного класса. В рассмотренном примере это относится к процедуре **F1**. Если в модуле описано несколько классов, то объекты этих классов взаимно видят элементы, описанные в их разделах **private**.
- Элементы, объявленные в классе в разделе **public**, видимы и доступны для объектов любых классов и для других модулей, в которых директивой **#include** включен заголовочный файл данного модуля. При этом из объектов того же класса, к ним можно обращаться непосредственно по имени, а из других объектов и процедур — только со ссылкой на объект данного класса. В рассмотренном примере это относится к переменной **Ch1** и процедуре **F2**.
- В классах, помимо обсуждавшихся ранее, могут быть еще разделы **protected** — защищенные. Элементы, объявленные в классе в разделе **protected**, видимы и доступны для любых объектов внутри данного модуля, а также для объектов классов — наследников данного класса в других модулях. Объекты из других модулей, классы которых не являются наследниками данного класса, защищенных элементов не видят.
- Элементы, объявленные внутри функции или блока (в рассмотренном примере это переменная **Ch4** объявленная внутри функции **TForm1::Button1Click**), являются локальными, т.е. они видимы и доступны только внутри данной функции или данного блока. При этом время жизни переменных, объявленных внутри функции или блока, определяется временем активности данного блока. Так переменная **Ch4** в нашем примере создается в момент вызова функции **TForm1::Button1Click** и уничтожается при завершении работы этой функции. Сделать локальную переменную существующей постоянно можно с помощью спецификации **static**.

- Переменные и функции, объявленные в головном файле проекта, являются глобальными для этого файла. Если требуется доступ к ним из других модулей, то для функций в них должны быть повторены их объявления, а для переменных — повторено объявление (без инициализации) со спецификацией **extern**.
- Если во внутреннем блоке объявлена переменная с тем же именем, что во внешнем блоке, или с тем же именем, что и глобальная переменная, то соответствующая внешняя или глобальная переменная в блоке не видна. В этом случае подучить доступ к одноименной глобальной переменной можно только с помощью унарной операции разрешения области действия «::».

Более подробные сведения об областях видимости переменных и функций вы можете найти в главе 12 в разделе 12.6.

1.5.5.5 Передача параметров в функции

Параметры в функции могут передаваться в основном двумя способами: *по значению* и *по ссылке*. То, что вы могли видеть в рассмотренном ранее примере — это передача параметра по значению. В этом случае в заголовке функции указывается имя параметра и его тип. Например:

```
void F1(char Ch);
```

В этом случае **Ch** — это локальное имя формального параметра, используемое только внутри данной функции. При вызове этой функции, который может иметь вид:

```
F1(Ch2);
```

в памяти создается временная переменная с именем **Ch**, и в нее копируется значение аргумента **Ch2**. На этом связь между **Ch** и **Ch2** разрывается. Вы можете изменить внутри функции значение **Ch**, но это никак не отразится на значении внешней переменной **Ch2**, указанной в вызове функции в качестве аргумента.

Такая передача параметров по значению имеет свои достоинства и недостатки. Достоинство заключается в том, что функция не может испортить переданный в нее аргумент. Это важно для надежной работы приложения и позволяет разрабатывать функции, не задумываясь о том, не использованы ли где-то в программе те же имена параметров. Но у передачи параметра по значению имеется и ряд недостатков. Во-первых, функция не может изменить значение аргумента, а иногда это очень желательно. Во-вторых, копирование значения аргумента требует дополнительных затрат времени и памяти для хранения копии. Если речь идет о какой-то переменной простого типа, это, конечно, не существенно. Но если, например, аргумент — это массив из тысячи элементов, то соображения затрат времени и памяти могут стать существенными. И в-третьих, после окончания работы функции временная переменная, хранившая значение параметра, уничтожается. Поэтому ее нельзя использовать, например, для накопления какой-то информации.

Для реализации второго способа передачи информации — по ссылке, перед именем параметра в заголовке функции должен быть записан символ амперсанта '&'. Например:

```
void F1(char &Ch);
```

В этом случае не происходит копирования значения аргумента в локальную, временную переменную в процедуре. Процедура реально работает не с параметром, а со ссылкой — указателем на место хранения аргумента в памяти. И любые изменения параметра **Ch**, произведенные в процедуре, в действительности относятся не к этому параметру, а к тому аргументу, который передан при вызове процедуры. Таким образом, передача параметра по ссылке позволяет возвращать информацию из функции в вызвавшую его внешнюю процедуру.

Более подробную информацию о передаче параметров в функции вы найдете в разделе 12.5.2 главы 12.

1.5.6 Работа с указателями на объекты

Основой объектно-ориентированного программирования являются объекты и классы. В предыдущих разделах мы уже рассмотрели тот минимум сведений о них, который необходим для начала продуктивной работы с C++Builder. В данном разделе мы рассмотрим более сложные и тонкие вопросы работы с объектами. Читатель, слабо знакомый или вовсе не знакомый с указателями, может без особых потерь пропустить пока этот раздел и сразу перейти к материалу главы 2. Это не помешает ему эффективно работать и создавать свои интересные приложения. А когда возникнет потребность разобраться в каких-то вопросах, связанных с объектами и указателями на них, можно, имея уже некоторый опыт, вернуться к этому разделу и разобраться в неясных деталях. Но автор все-таки решил вынести эти вопросы в начало книги, поскольку в дальнейшем изложении придется достаточно часто ссылаться на рассмотренные ниже приемы распознавания объектов и работы с ними. Впрочем, если нет особого желания углубляться в детали, можно рассматривать эти приемы просто как некую данность и копировать их в свое приложение, когда потребуется решать соответствующие задачи.

1.5.6.1 Указатели на объекты

Указатель на объект — это переменная, в которой хранится ссылка на объект некоторого класса, т.е. на место в памяти, где размещен объект. Объявление (создание) указателя на объект некоторого класса (типа) производится оператором:

```
<тип> *<указатель>;
```

Например, следующее объявление создает указатель на объект класса **TLabel** (метку):

```
TLabel *Lab;
```

Создается не сам объект, а только указатель на любой объект данного типа. В момент его создания указатель инициализируется нулем. Ноль не может ассоциироваться ни с каким объектом в памяти. Поэтому определить, занесена в указатель ссылка на конкретный объект, или нет, можно, например, оператором:

```
if (Lab == 0) ...;
```

Здесь многоточием обозначены некие действия, которые надо делать при отсутствии ссылки.

В C++ предопределена константа **NULL**, которая эквивалентна нулевому указателю. Поэтому приведенный выше оператор эквивалентен следующему:

```
if (Lab == NULL) ...;
```

В дальнейшем указателю можно присвоить ссылку на любой объект соответствующего класса простым присваиванием. Например:

```
Lab = Label1;
```

Тогда указатель **Lab** становится как бы псевдонимом объекта **Label1**. Оба указателя: и **Lab**, и **Label1** ссылаются на один и тот же объект. Например, **Label1->Caption** и **Lab->Caption** ссылаются на надпись одной и той же метки.

В качестве типа объекта, на который ссылается указатель, можно задать **void**. Например:

```
void *Lab;
```

Такой указатель на **void** можно рассматривать как указатель на объект любого типа. В дальнейшем этому указателю можно задать простым присваиванием ссылку на объект любого типа. Но разыменование такого указателя требует применения явного приведения типов, поскольку компилятор не знает, на объект какого типа в действительности ссылается указатель. Поэтому для него нельзя, напри-

мер, после присваивания ему ссылки на метку **Label1** (как в приведенном ранее примере) написать просто **Lab->Caption**. Для ссылки на надпись **Caption** через этот указатель надо писать **((TLabel *)Lab)->Caption**, то есть явным образом приводить тип указателя **Lab** к типу «указатель на объект класса **TLabel**».

Можно создавать ссылку на объект с помощью указателя не на истинный класс объекта, а на один из классов, которым наследует класс данного объекта. Дело в том, что любой объект может рассматриваться не только как объект своего класса, но и как объект любого класса-предка. Это в общем достаточно естественно для обычного понимания объектов в реальном мире. Так любой автомобиль может рассматриваться не только как объект автомобилей данной марки, например, «Жигули», но и как один из объектов более общих классов — автомобили, средства передвижения и т.д. Так же и объект в C++ может рассматриваться как объект любого из классов предков.

Например, универсальным указателем на любой компонент может быть указатель на класс **TControl** — базовый класс всех компонентов:

```
TControl *Contr;
```

Такому указателю можно непосредственно присваивать ссылку на любой компонент. Например,

```
Contr = Label1;
```

При таком присваивании компилятор сам производит необходимое приведение типов.

Но с этим указателем **Contr** уже нельзя работать непосредственно как с указателем на метку. Для него известны только свойства, объявленные в классе **TControl**. Это такие общие свойства всех компонентов, как, например, **Name** — имя. Непосредственная ссылка на специфические свойства классов — наследников невозможна. Например, попытка написать код **Contr->Caption** вызовет сообщение компилятора об ошибке с текстом: «'Controls::TControl::Caption' is not accessible.», смысл которого заключается в том, что свойство **Caption** в классе **TControl** недоступно. Поэтому для доступа к методам и свойствам, отсутствующим в классе **TControl**, надо осуществлять явное приведение типа указателя, например:

```
((TLabel *)Contr)->Caption
```

Этот код как бы говорит компилятору: «Рассматривай **Contr** как ссылку на класс **TLabel**». И тогда никаких сообщений об ошибках не возникает.

Причина, по которой в ряде случаев для хранения ссылок на объекты используются указатели на объекты базовых классов, заключается в удобстве групповой обработки объектов разных классов с помощью общих для них методов или для задания значений общих для них свойств. Этот вопрос будет подробнее рассмотрен в следующем разделе.

Выше рассматривалось присваивание указателю ссылки на уже существующий объект. Но создание нового объекта тоже всегда связано с созданием указателя на него. Для создания объекта применяется операция **new**. В ней после ключевого слова **new** указывается класс компонента. Если объект является компонентом, то после класса компонента в скобках указывается компонент — хозяин (**Owner**). Кроме того после создания компонента обязательно надо указать его родителя (**Parent**) — компонент, на котором размещается новый компонент. Пока не задан родитель, компонент нельзя увидеть.

Рассмотрим следующий код:

```
TLabel *Lab1 = new TLabel (Form1);
Lab1->Parent = Form1;
Lab1->Caption = "Это новая метка";
```

Первый из этих операторов создает указатель **Lab1** на объект-метку типа **TLabel** и создает с помощью **new** сам объект-метку с хозяином **Form1**. Последующие операторы используют указатель **Lab1** для задания свойств метки: размещают ее на форме **Form1** и задают на ней надпись «Это новая метка»:

Поскольку никакие другие свойства метки не заданы, они будут установлены конструктором класса **TLabel** с умолчанием. Этот конструктор неявным образом вызывается операцией **new**.

1.5.6.2 Идентификация объекта неизвестного класса

В предыдущем разделе было рассмотрено объявление указателей на объекты и было показано, что тип такого указателя может определяться не обязательно классом конкретного объекта, но и любым классом-предком. В C++Builder это используется достаточно широко. Например, во все обработчики событий передается в качестве параметра **Sender** — указатель на объект, в котором произошло событие. Зачем нужен этот параметр? Конечно, если вы пишете обработчик какого-то события в конкретном компоненте, например, пишете для кнопки **Button1** обработчик события **OnClick**, которое наступает при щелчке на ней мыши, то параметр **Sender** вам не нужен. Вы и без этого параметра знаете, что событие произошло именно в кнопке **Button1**. Но часто для разных компонентов нужна идентичная реакция на идентичные события. В этих случаях писать отдельные одинаковые обработчики для разных компонентов нерационально. Можно ограничиться одним обработчиком для всех этих компонентов. Это обеспечит существенно более компактный код, его будет проще отлаживать, да и размер загрузочного модуля вашей программы будет меньше.

В этой книге вы найдете немало таких примеров. Приведем в чисто описательном плане некоторые из них. В главе 4 в разделах 4.1.8 и 4.3.2.2 описана задача разработки окон редактирования таких, чтобы при нажатии пользователем клавиши **Enter** фокус передавался бы следующему окну редактирования. Обработка нажатия клавиш во всех таких окнах может быть сделана одним обработчиком, что, конечно, лучше, чем писать одинаковые обработчики для каждого окна. В той же главе в разделе 4.2.5 описана задача синхронного масштабирования всех оконных компонентов, содержащихся в некотором контейнере. С помощью указателя на базовый класс **TWinControl** всех оконных компонентов эта задача решается простым циклом, причем изменяются размеры всех оконных компонентов, независимо от того, к какому классу каждый из них принадлежит. Это много эффективнее, чем писать отдельный оператор для каждого компонента. В разделе 4.4.1 описывается техника **Drag&Drop** — перетаскивание информации об объектах. В этом примере также обобщенный подход позволяет обойтись одним обработчиком события для разных объектов и даже разных классов объектов. В главе 3 в разделе 3.8.7 описана организация поиска фрагмента текста в диалогах **Найти** и **Заменить**. Для этих разных классов компонентов процедура поиска одна и та же, так что ее можно осуществлять одним обработчиком.

Подобных примеров в книге будет много. Причем во многих случаях вопрос не только в эффективности кода. Если вы в процессе проектирования добавляете в приложение какие-то новые компоненты, то в вашем коде, написанном в общем виде, ничего не меняется. Новые компоненты автоматически обрабатываются теми же обработчиками. А если бы вы не использовали такой обобщенный подход, вам пришлось бы для каждого нового компонента писать новый код.

Не останавливаясь пока на деталях перечисленных примеров, рассмотрим общие подходы к подобным задачам. Параметр **Sender**, используемый в ряде этих примеров, объявлен в заголовках функций в C++Builder как **TObject *Sender**, т.е. как указатель на объект типа **TObject**. Класс **TObject**, как вы можете увидеть в разделе 16.4 главы 16, является базовым классом всех компонентов в C++Builder. Но в нем не объявлено никаких свойств, которые можно было бы использовать в

обработчике события. Поэтому при обращении к каким-то свойствам объектов вам надо явным образом осуществлять приведение типа параметра **Sender** к тому классу, в котором требуемые свойства объявлены. Пусть, например, вы пишете обработчик, который должен в качестве надписи (свойство **Caption**) метки **Label1** вывести текст: «Произошло событие в компоненте ...». Имя компонента, которое вам надо включать в эту надпись, содержится в свойстве **Name**. Но это свойство появляется только начиная с класса **TComponent** (см. главу 16 раздел 16.4) Значит именно к этому классу вам надо привести тип параметра **Sender**. Тогда соответствующий оператор будет иметь вид:

```
Label1->Caption = "Произошло событие в компоненте " +
                  ((TComponent *)Sender)->Name;
```

Выражение **((TComponent *)Sender)** является приведением типа параметра **Sender** к типу указателя на объект класса **TComponent**. Только в этом классе и в его потомках появляется свойство **Name**, которое вам нужно. Приведенный оператор будет работать для любых компонентов: окон, меток, кнопок и т.д., поскольку классы всех компонентов являются производными от **TComponent**.

Если подобное приведение типов требуется во многих операторах вашей функции, то для сокращения записи можно один раз определить указатель на объект **Sender** как указатель на требуемый класс, а затем во всех операторах использовать его. Это сделано, например, в следующем коде:

```
TComponent *Obj = (TComponent *)Sender;
...
Label1->Caption="Произошло событие в компоненте " + Obj->Name;
```

Первый из этих операторов объявляет переменную **Obj** как указатель на объект класса **TComponent** и с помощью явного приведения типа присваивает этому указателю ссылку на тот объект, на который указывает **Sender**. После этого переменную **Obj** можно везде использовать как указатель на этот объект.

Аналогично, если вы хотите применить к параметру **Sender** некоторый метод, объявленный в классах-наследниках, вы должны привести тип указателя к тому классу, где этот метод имеется. Например, если вы хотите увеличить масштаб оконного компонента, указателем на который является **Sender**, вы должны привести его тип к указателю на объект класса **TWinControl** (или одного из производных от него классов), так как только начиная с базового класса всех оконных компонентов **TWinControl** объявлен требуемый вам метод **ScaleBy**. Соответствующий оператор будет иметь вид:

```
((TWinControl *)Sender)->ScaleBy(11,10);
```

В ряде случаев требуется определить истинный класс объекта, на который указывает параметр **Sender**. Это можно сделать с помощью метода **ClassName**, объявленного в классе **TObject** как

```
ShortString __fastcall ClassName();
```

Функция **ClassName** возвращает строку типа **ShortString**, содержащую истинный класс объекта. Например, оператор

```
Label1->Caption = Sender->ClassName();
```

может выдать текст «**TButton**», если **Sender** указывает на кнопку типа **TButton**.

Функцию **ClassName** можно использовать для выполнения каких-то действий с объектами только одного конкретного класса. Например, оператор

```
if (String(Sender->ClassName()) == "TLabel")
...;
```

обеспечивает выполнение неких действий (обозначенных многоточием) только для объектов класса **TLabel**.

Для тех же целей может использоваться еще одна функция — **ClassNameIs**, объявленная в классе **TObject** как:

```
bool __fastcall ClassNameIs(const AnsiString string);
```

Эта функция возвращает **true**, если класс объекта совпадает с заданным параметром **string**. При использовании этой функции приведенный выше пример приобретает вид:

```
if (Sender->ClassNameIs("TLabel"))  
    ...;
```

В приведенных ранее примерах использования свойств и методов класса, к которому приводится указатель на класс-предшественник, вас может подстерегать некая опасность. Выше был приведен пример масштабирования компонента с использованием метода **ScaleBy**, объявленного в классе **TWinControl**. Но если истинный класс объекта, на который указывает **Sender**, окажется не потомком класса **TWinControl** (например, меткой **TLabel**, которая не наследует **TWinControl**), то метод **ScaleBy** не сработает. Еще более неприятные и непредсказуемые результаты получатся, если вы обратитесь к свойству, отсутствующему у компонента.

Поэтому, если нет уверенности, что применяемый метод или свойство имеется в обрабатываемом объекте, надо предварительно проверить, является ли класс объекта потомком того класса, в котором требуемый метод или свойство объявлены. Например, прежде, чем применять метод **ScaleBy**, надо убедиться, что класс объекта является потомком **TWinControl**.

Для этих целей можно воспользоваться методом **InheritsFrom**, объявленным в классе **TObject** и, следовательно, имеющимся в любых компонентах. Объявление этого метода:

```
bool __fastcall InheritsFrom(TClass aClass);
```

Метод возвращает **true**, если класс данного объекта является потомком класса **aClass**, указываемого как параметр метода. Этот параметр имеет тип **TClass**, который может создаваться операцией **__classid**:

```
__classid(classType)
```

Аргументом этой операции является обычное имя класса, например, **TWinControl**.

Таким образом, проверка, является ли класс объекта, на который указывает **Sender**, потомком **TWinControl**, может осуществляться оператором:

```
if (Sender->InheritsFrom(__classid(TWinControl)))  
    ...;
```

С учетом этого приведенный ранее пример масштабирования методом **ScaleBy** оконных компонентов более грамотно должен осуществляться следующим оператором:

```
if (Sender->InheritsFrom(__classid(TWinControl)))  
    ((TWinControl *)Sender)->ScaleBy(11,10);
```

Еще одна функция — **ClassParent**, объявленная в классе **TObject**, возвращает класс, являющийся непосредственным предком класса данного объекта. Функция объявлена как:

```
TClass __fastcall ClassParent()
```

Если данный класс не имеет предшественников (т.е. это класс **TObject**), то возвращается **NULL**.

Функция **ClassParent**, используемая в цикле, позволяет восстановить всю иерархию класса объекта. Следующий код заносит в список строки, перечисляющие все классы, встречающиеся на пути по дереву классов от **TObject** до класса, на который указывает параметр **Sender**.


```
TClass ClassRef= Sender->ClassType();
ListBox1->Clear();
while(ClassRef != NULL)
{
    ListBox1->Items->Add(ClassRef->ClassName());
    ClassRef = ClassRef->ClassParent();
}
```

Так, если **Sender** указывает на объект типа **TButton**, то в списке **ListBox1** окажется текст:

```
TButton
TButtonControl
TWinControl
TControl
TComponent
TPersistent
TObject
```

На этом мы закончим наше первое знакомство с C++. Это знакомство будет продолжаться на всем протяжении книги, а для более глубокого изучения вы всегда можете воспользоваться главами 12 и 13 справочной части книги. Но и сейчас вы уже готовы начать работать непосредственно с C++Builder и создавать свои первые приложения. Этим вы и займетесь в следующей главе.

Глава 2

Система визуального объектно-ориентированного программирования C++Builder

2.1 Что может C++Builder 5

C++Builder 5 — мощная система визуального объектно-ориентированного проектирования. Он сам и поставляемые с ним программные продукты позволяют решать следующий круг задач:

- Быстро создавать профессионально выглядящий оконный интерфейс для любых приложений даже начинающим программистам. Интерфейс удовлетворяет всем требованиям Windows, настраивается на используемую систему, поскольку использует многие функции, процедуры, библиотеки Windows.
- Создавать приложения любой сложности и любого назначения: офисные, бухгалтерские, инженерные, информационно-поисковые — никаких преград перед C++Builder и лежащим в его основе языком C++ нет.
- Создавать современный пользовательский интерфейс для любых ранее разработанных программ DOS и Windows. Нередко в учреждении или фирме существуют и успешно эксплуатируются прикладные программы, разработанные в разное время, разными коллективами, для разных операционных систем. С помощью C++Builder эти приложения можно снабдить современным удобным оконным интерфейсом, объединить разрозненные приложения в единую систему, обеспечить их стилистическое единство, наладить обмен информации между приложениями.
- Создавать свои библиотеки .DLL компонентов, форм, функций, которые затем можно использовать из других языков программирования.
- Создавать мощные системы работы с локальными и удаленными базами данных любых типов. Подход, используемый в C++Builder, позволяет получить доступ к базам, созданным на любой платформе: InterBase, Microsoft Access, FoxPro, Paradox, dBase, Sybase, Microsoft SQL, Oracle и др.
- Создавать базы данных многих типов с помощью инструментария C++Builder.
- Автономно отлаживать приложения работы с базами данных на локальном сервере InterBase, поставляемом вместе с C++Builder, с последующим выходом в сеть.
- Формировать и печатать из приложения сложные отчеты, включающие таблицы, графики и т.п. самого различного назначения.
- Связываться из своего приложения с такими продуктами Microsoft, как Word, Excel и другие, используя все их богатейшие возможности.
- Создавать системы помощи (Help), как для своих приложений, так и для любых других, с которыми, в частности, можно работать просто через Windows.
- Создавать профессиональные программы установки приложений Windows, учитывающие всю специфику и все требования Windows. В частности, для

этого можно использовать поставляемую вместе с C++Builder программу InstallShield Express.

- и многое другое.

2.2 Интегрированная Среда Разработки (ИСР) C++Builder

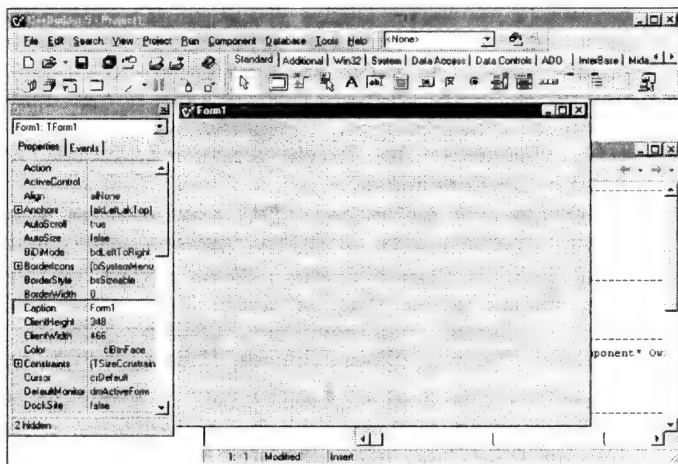
2.2.1 Общий вид окна ИСР

Интегрированная Среда Разработки (Integrated Development Environment — IDE, в дальнейшем мы будем использовать для нее аббревиатуру ИСР) — это среда, в которой есть все необходимое для проектирования, запуска и тестирования приложений и где все нацелено на облегчение процесса создания программ. ИСР интегрирует в себе редактор кодов, отладчик, инструментальные панели, редактор изображений, инструментарий баз данных — все, с чем приходится работать. Эта интеграция предоставляет разработчику гармоничный набор инструментов, дополняющих друг друга. Более того, как вы увидите в дальнейшем, вам предоставлена возможность расширять меню ИСР, включая в него необходимые вам дополнительные программы, в том числе и собственные. Результатом является удобная для вас среда быстрой разработки сложных прикладных программ.

Запустите C++Builder, выбрав пиктограмму C++Builder 5 в разделе меню Windows Пуск | Программы. Когда вы щелкнете на пиктограмме C++Builder, перед вами откроется основное окно Интегрированной Среды Разработки. Его вид представлен на рис. 2.1.

Рис. 2.1

Основное окно
Интегрированной Среды
Разработки C++Builder 5



В верхней части окна ИСР вы видите *полосу главного меню*. Ее состав частично зависит от варианта C++Builder, с которым вы работаете. На рис. 2.1 приведен вид окна для варианта Enterprise.

Ниже полосы главного меню расположены две *инструментальные панели*. Левая панель (состоящая в свою очередь из нескольких панелей) содержит два ряда *быстрых кнопок*, дублирующих некоторые наиболее часто используемые команды меню. Правая панель содержит *палитру компонентов* библиотеки визуальных компонентов (Visual Component Library — VCL). В дальнейшем мы для краткости будем называть библиотеку визуальных компонентов просто библиотекой, благо это ближе к истине, так как в ней содержатся и визуальные, и не визу-

альные компоненты. Палитра компонентов содержит ряд страниц, закладки которых видны в ее верхней части.

Правее полосы главного меню размещена еще одна небольшая инструментальная панель, содержащая выпадающий список и две быстрые кнопки. Это панель сохранения и выбора различных конфигураций окна ИСР, которые вы сами можете создавать и запоминать.

В основном поле окна вы можете видеть слева окно Инспектора Объектов (Object Inspector), с помощью которого вы в дальнейшем будете задавать свойства компонентов и обработчики событий. Правее вы можете видеть окно пустой формы, готовой для переноса на нее компонентов. Под ним расположено окно Редактора Кодов. Обычно оно при первом взгляде на экран невидимо, так как его размер равен размеру формы и окно Редактора Кодов практически полностью перекрывается окном формы. На рис. 2.1 это окно немного сдвинуто и выглядывает из под окна формы.

Рассмотрим теперь основные элементы окна ИСР.

2.2.2 Полоса главного меню и всплывающие меню

В главе 14 в разделе 14.1 дается описание всех разделов меню. Кроме того в дальнейшем при обсуждении различных проектных операций мы еще будем подробно рассматривать функции многих разделов меню. А пока просто дадим краткий обзор основных разделов.

Разделы меню File (файл) позволяют создать новый проект, новую форму, открыть ранее созданный проект или форму, сохранить проекты или формы в файлах с заданными именами.

Разделы меню Edit (правка, редактирование) позволяют выполнять обычные для приложений Windows операции обмена с буфером Clipboard, а также дают возможность выравнивать группы размещенных на форме компонентов по размерам и местоположению.

Разделы меню Search (поиск) позволяют осуществлять поиск и контекстные замены в коде приложения, которые свойственны большинству известных текстовых редакторов.

Разделы меню View (просмотр) позволяют вызывать на экран различные окна, необходимые для проектирования.

Разделы меню Project (проект) позволяют добавлять и убирать из проекта формы, задавать опции проекта, компилировать проект без его выполнения и делать много других полезных операций.

Меню Run (выполнение) дает возможность выполнять проект в нормальном или отладочном режимах, продвигаясь по шагам, останавливаясь в указанных точках кода, просматривая значения переменных и т.д.

Меню Component (компонент) позволяет создавать и устанавливать новые компоненты, конфигурировать палитру компонентов, работать с пакетами.

Разделы меню Database (база данных) позволяют использовать инструментальный для работы с базами данных.

Меню Tools (инструментарий) включает ряд разделов, позволяющих настраивать ИСР и выполнять различные вспомогательные программы, например, вызывать Редактор Изображений (Image Editor), работать с программами, конфигурирующими базы данных и т.д. Кроме того, в это меню вы можете сами включить любые разделы, вызывающие те или иные приложения, и таким образом расширить возможности главного меню C++Builder, приспособив его для своих задач (подробнее см. в разделе 14.2.4).

Меню Help (справка) содержит разделы, помогающие работать со встроенной в C++Builder справочной системой.

Мы рассмотрели основные меню, входящие в полосу главного меню. Но помимо главного меню в C++Builder имеется система *контекстных всплывающих меню*, которые появляются, если пользователь поместил курсор мыши в том или ином окне или на том или ином компоненте и щелкнул правой кнопкой мыши. Большинство разделов этих контекстных меню дублируют основные разделы главного меню. Однако, во всплывающих меню в ряде случаев имеются разделы, отсутствующие в главном меню. И до многих инструментов, используемых для работы с некоторыми компонентами, можно добраться только через всплывающие меню. Так что почаще пробуйте в процессе работы щелкать правой кнопкой мыши. Это ускорит выполнение многих проектных операций.

2.2.3 Быстрые кнопки

Инструментальные панели быстрых кнопок для C++Builder 5 представлены на рис. 2.2. Как будет видно позднее, фактически это не две, а пять панелей. Назначение размещенных на них быстрых кнопок можно узнать из ярлычков, появляющихся, если вы поместите курсор мыши над соответствующей кнопкой и на некоторое время задержите его. В таблице 2.1 приведены пиктограммы этих кнопок, соответствующие им команды меню и «горячие» клавиши, а также краткие пояснения.

Рис. 2.2
Инструментальные панели в C++Builder 5: основные (а) и панель выбора конфигурации окна (б)

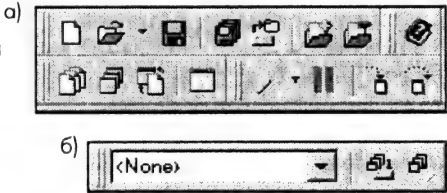


Таблица 2.1. Быстрые кнопки

Пиктограммы	Команда меню / «горячие» клавиши	Пояснение команды
	File New	Открыть проект или модуль из Депозитария
	File Open File Reopen	Открыть файл проекта, модуля, пакета. Кнопка со стрелкой справа от основного изображения соответствует команде Reopen, позволяющей открыть файл из списка недавно использовавшихся
	File Save (Ctrl — S)	Сохранить файл модуля, с которым в данный момент идет работа
	File Save All	Сохранить все (все файлы модулей и файл проекта)
	File Open Project (Ctrl — F11)	Открыть файл проекта
	Project Add to Project (Shift — F11)	Добавить файл в проект
	Project Remove from Project	Удалить файл из проекта

Пиктограммы	Команда меню / «горячие» клавиши	Пояснение команды
	Help C++Builder Help	Вызов страницы Содержание встроенной справки
	View Units (Ctrl — F12)	Переключиться на просмотр текста файла модуля, выбираемого из списка
	View Forms (Shift — F12)	Переключение на просмотр формы, выбираемой из списка
	View Toggle Form/Unit (F12)	Переключение между формой и соответствующим ей файлом модуля
	File New Form	Включить в проект новую форму
	Run Run (F9)	Выполнить приложение. Кнопочка со стрелкой справа от основного изображения позволяет выбрать выполняемый файл, если вы работаете с группой приложений
	Run Program Pause	Пауза выполнения приложения и просмотр информации CPU. Кнопка и соответствующий раздел меню доступны только во время выполнения приложения
	Run Trace Into (F7)	Пошаговое выполнение программы с заходом в функции
	Run Step Over (F8)	Пошаговое выполнение программы без захода в функции
	View Desktops Save Desktop	Сохранение текущей конфигурации окна
	View Desktops Set Debug Desktop	Установка конфигурации окна при отладке

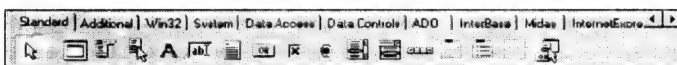
На рис. 2.2 и в таблице 2.1 приведен стандартный состав инструментальных панелей быстрых кнопок. Однако, в C++Builder 5 вам предоставляются широкие возможности настроить панели по своему усмотрению, добавить в них какие-то быстрые кнопки для часто применяемых вами команд, убрать кнопки, которыми вы редко пользуетесь, сделать некоторые из инструментальных панелей невидимыми. Настройка инструментальных панелей быстрых кнопок рассмотрена в главе 14 в разделе 14.2.1.

2.2.4 Палитра компонентов

Палитра компонентов (рис. 2.3) — это витрина библиотеки визуальных компонентов (Visual Component Library — VCL). Она позволяет сгруппировать компоненты в соответствии с их смыслом и назначением. Эти группы или *страницы* снабжены закладками. Вы можете изменять комплектацию страниц, вводить новые страницы, переставлять их, вносить на страницы разработанные вами шаблоны и компоненты и т.д.

Рис. 2.3


Палитра компонентов



По умолчанию в палитре C++Builder 5 имеются страницы:

Standard	Стандартная, содержащая наиболее часто используемые компоненты
Additional	Дополнительная, являющаяся дополнением стандартной
Win32	32-битные компоненты в стиле Windows 95/98 и NT
System	Системная, содержащая такие компоненты, как таймеры, плееры и ряд других
Data Access	Доступ к данным через Borland Database Engine (BDE)
Data Controls	Управление данными
ADO	Связь с базами данных через Active Data Objects (ADO) — множество компонентов ActiveX, использующих для доступа к информации баз данных Microsoft OLE DB
InterBase	Прямая связь с InterBase, минуя Borland Database Engine (BDE) и Active Data Objects (ADO)
Midas	Построение приложений баз данных с параллельными потоками
InternetExpress	Построение приложений InternetExpress — одновременно приложений сервера Web и клиента баз данных с параллельными потоками
Internet	Компоненты для приложений, работающих с Интернет
FastNet	Различные протоколы доступа к Интернет
Decision Cube	Многомерный анализ данных
Qreport	Быстрая подготовка отчетов
Dialogs	Стандартные системные диалоги Windows
Win 3.1	Компоненты в стиле Windows 3.x (для обратной совместимости)
Samples	Образцы, различные интересные, но не до конца документированные компоненты
ActiveX	Активные элементы ActiveX
Servers	Оболочки VCL для распространенных серверов COM

Поскольку число страниц в C++Builder 5 велико и не все закладки видны на экране одновременно, в правой части палитры компонентов имеются две кнопки со стрелками, направленными влево и вправо. Эти кнопки позволяют перемещать отображаемую на экране часть палитры.

Чтобы перенести компонент на форму, надо открыть соответствующую страницу библиотеки и указать курсором мыши необходимый компонент. При этом кнопка-указатель , размещенная в левой части палитры компонентов, приобретет вид не нажатой кнопки. Это значит, что вы находитесь в состоянии, когда собираетесь поместить компонент на форму. Если вы нажмете эту кнопку, это будет означать, что вы отказались от размещения выбранного компонента.

Поместить выбранный в палитре компонент на форму очень просто — надо сделать щелчок мышью в нужном месте формы. Есть и другой способ поместить компонент на форму — достаточно сделать двойной щелчок на пиктограмме компонента в палитре, и он автоматически разместится в центре вашей формы. Если вы выбрали компонент, а затем изменили ваше намерение размещать его, вам дос-

таточно нажать кнопку указателя. Это прервет процесс размещения компонента и программа вернется в нормальный режим, в котором вы можете выбирать другой компонент или выполнять какую-то команду.

Имена компонентов, соответствующих той или иной пиктограмме, вы можете узнать из ярлычка, появляющегося, если вы задержите над этой пиктограммой курсор мыши. Если вы выберете в палитре компонент и нажмете клавишу F1, то вам будет показана справка по типу данного компонента. Тут надо сразу сделать одно замечание. Имена на ярлычках выглядят, например, так: **MainMenu**, **Button** и т.д. Однако, в C++Builder все имена классов в действительности начинаются с символа «Т», например, **TMainMenu**, **TButton**. Под такими именами вы можете найти описания соответствующих компонентов во встроенной в C++Builder справочной системе.

2.2.5 Окно формы

Основой почти всех приложений C++Builder является форма. Ее можно понимать как типичное окно Windows. Форма является основой, на которой размещаются другие компоненты.

Форма имеет те же свойства, что присущи другим окнам Windows 95/98. Она имеет управляющее меню в верхнем левом углу, полосу заголовка, занимающую верхнюю часть окна, кнопки разворачивания, свертывания и закрытия окна в верхнем правом углу. Можно изменить вид окна, убрав в нем какие-то кнопки или всю полосу заголовка, сделав его окном с неизменяемыми размерами и т.п. О том, как это сделать, вы узнаете в разделе 4.1.3 главы 4.

Во время проектирования форма покрыта сеткой из точек. В узлах этой сетки размещаются те компоненты, которые вы помещаете на форму. Во время выполнения приложения эта сетка, конечно, не видна.

В некоторых случаях при разработке какого-то модуля форма может оказаться вообще ненужной. Но обычно вся работа в C++Builder проводится именно на форме.

Когда вы поместили на форме какие-то компоненты, вы можете получить по ним контекстную справку. Для этого выделите интересующий вас компонент и нажмете клавишу F1. Если вы щелкнете на самой форме и нажмете клавишу F1, вам будет показана справка по классу формы.

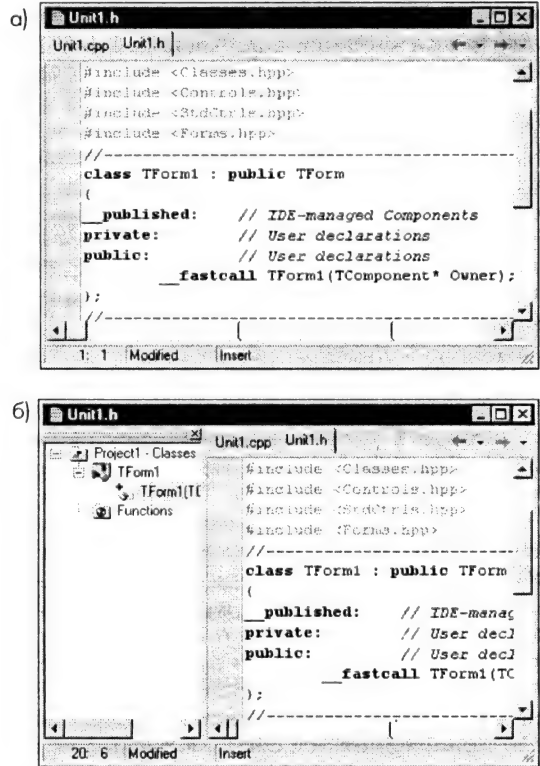
2.2.6 Окно Редактора Кода

Одной из наиболее важных частей среды C++Builder является окно Редактора Кода, показанное на рис. 2.4 а. В действительности, если вы откроете в первый раз это окно в C++Builder 5, оно может выглядеть несколько иначе (рис 2.4 б) и включать в себя слева еще одно встроенное окно — окно Исследователя Классов (ClassExplorer), подробно рассмотренное в разделе 2.5.3.2. Но об этом будет сказано позднее и во многих случаях вы просто можете закрыть это дополнительное окно, щелкнув на кнопке в его правом верхнем углу, или задать опции среды проектирования, отменяющие по умолчанию появление окна ClassExplorer (см. раздел 14.2.7 главы 14).

Редактор Кода является полноценным программным редактором. Его можно настраивать на различный стиль работы, который вам более привычен. В редакторе применяется выделением цветом и шрифтом синтаксических элементов. Жирным шрифтом выделяются ключевые слова C++ (на рис. 2.4 вы видите выделенные слова **class**, **public**, **__published** и другие). Зеленым цветом выделяются директивы препроцессора (на рис. 2.4 это директивы **#include**). Синим курсивом выделяются комментарии (на рис. 2.4 это тексты *«// IDE-managed Components»* и *«// User declarations»*).

Рис. 2.4

Окно Редактора Кода без встроенного окна ClassExplorer (а) и со встроенным окном (б)



В заголовке окна Редактора Кода отображается имя текущего файла, того, с текстом которого вы работаете. В приложениях C++Builder часто приходится работать с несколькими файлами. В частности, обычно кроме файла реализации модуля **.cpp** вам нужен еще заголовочный файл модуля **.h** (см. в главе 1 разделы 1.5.2 и 1.5.4). Вы можете загрузить заголовочный файл в Редактор Кода, щелкнув в окне редактора правой кнопкой мыши и выбрав из всплывшего (контекстного) меню команду **Open Source/Header File**. Если в этот момент вы находились в окне Редактора Кода на странице с текстом файла реализации модуля, то в Редактор Кода загрузится заголовочный файл вашего модуля. На рис. 2.4 работа идет именно с заголовочным файлом. В верхней части окна вы можете видеть закладки или ярлычки, указывающие текущую страницу и помогающие переходить от одного файла к другому. Если какой-то из открытых файлов вам больше не нужен, вы можете закрыть его страницу в Редакторе Кода, выбрав в контекстном меню команду **Close Page**. Вы можете также открыть дополнительное окно Редактора Кода (командой **View | New Edit Window** или щелкнув в окне Редактора Кода правой кнопкой мыши и выбрав аналогичную команду из всплывшего меню) и одновременно работать с несколькими модулями или с разными фрагментами одного модуля.

В нижней части окна Редактора Кода вы можете видеть типичную для текстовых редакторов строку состояния. В самой левой ее позиции находится индикатор строки и колонки. Правее расположен индикатор модификации, который словом «**Modified**» показывает, что код, который вы видите в окне, изменен и не совпадает с тем, который хранится на диске. Третий элемент строки состояния — стандартный большинства редакторов индикатор режима вставки.

В окно Редактора Кода, как и в другие окна C++Builder, встроена контекстная справка. Чтобы получить справку по какому-то слову кода (ключевому слову, на-

писанному имени функции и т.п.) достаточно установить курсор на это слово и нажать клавишу F1. Вам будет показана соответствующая тема справки.

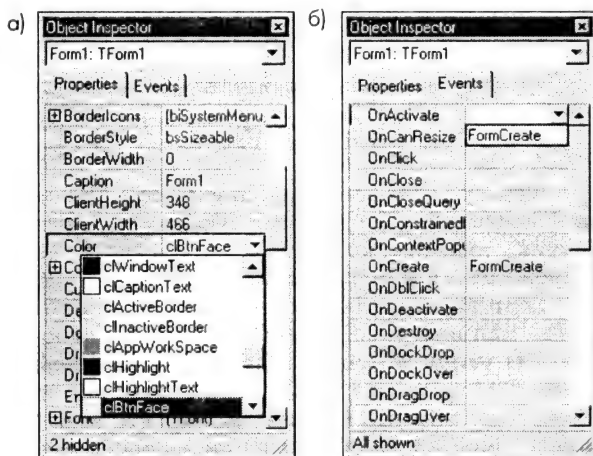
2.2.7 Инспектор Объектов

Инспектор Объектов (Object Inspector) обеспечивает простой и удобный интерфейс для изменения свойств объектов C++Builder и управления событиями, на которые реагирует объект.

Окно Инспектора Объектов (рис. 2.5) имеет две страницы. Выше их имеется выпадающий список всех компонентов, размещенных на форме. В нем вы можете выбрать тот компонент, свойства и события которого вас интересуют.

Рис. 2.5

Страница свойств (а) и страница событий (б) Инспектора Объектов



Страница свойств (Properties) Инспектора Объектов (см. рис. 2.5 а), показывает свойства того объекта, который в данный момент выделен вами. Щелкните на окне пустой формы и на странице свойств Инспектора Объектов вы сможете увидеть свойства формы (они показаны на рис. 2.5 а). Вы можете изменять эти свойства. Например, измените свойство **Caption** (надпись) вашей формы, написав в нем «Моя форма», и вы увидите, что эта надпись появится в полосе заголовка вашей формы.

Если щелкнуть на некоторых свойствах, например, на свойстве **Color** (цвет), то справа от имени свойства откроется окно выпадающего списка. Нажав в нем на кнопку со стрелкой вниз, вы можете увидеть список возможных значений свойства (см. рис. 2.5 а). Например, смените значение свойства **Color** с принятого по умолчанию **clBtnFace** (цвет поверхности кнопок) на **clWindow** (цвет окна). Вы увидите, что поверхность формы изменит свой цвет.

Рядом с некоторыми свойствами вы можете видеть знак плюс (см., например, свойство **BorderIcons** на рис. 2.5 а). Это означает, что данное свойство является объектом, который в свою очередь имеет ряд свойств.

Найдите, например, свойство **Font** (шрифт). Рядом с ним вы увидите знак плюс. Щелкните на этом плюсе или сделайте двойной щелчок на свойстве **Font**. Вы увидите, что откроется таблица таких свойств, как **Color** (цвет), **Height** (высота), **Name** (имя шрифта) и др. Среди них вы увидите свойство **Style** (стиль), около которого тоже имеется знак плюс. Щелчок на этом плюсе или двойной щелчок на этом свойстве раскроет дополнительный список подсвойств, в котором вы можете, например, установить в **true** свойство **fsBold** (жирный). Кстати, для смены **true** на **false** и обратно в подобных булевых свойствах не обязательно выбирать значение

из выпадающего списка. Достаточно сделать двойной щелчок на значении свойства, и оно изменится. После того, как вы просмотрели или изменили подсвойства, вы можете опять сделать двойной щелчок на головном свойстве или щелчок на знаке минус около него, и список подсвойств свернется.

Страница событий (Events) составляет вторую часть Инспектора Объектов (см. рис. 2.5 б). На ней указаны все события, на которые может реагировать выбранный объект. Например, если вам надо выполнить какие-то действия в момент создания формы (обычно это различные операции настройки), то вы должны выделить событие **OnCreate**. Рядом с именем этого события откроется окно с выпадающим списком. Если вы уже написали в своем приложении какие-то обработчики событий и хотите при событии **OnCreate** использовать один из них, вы можете выбрать необходимый обработчик из выпадающего списка. Если же вам надо написать новый обработчик, то сделайте двойной щелчок на пустом окне списка.

Вы попадете в окно Редактора Кода, в котором увидите текст:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
}
```

Курсор будет расположен в пустой строке между строками с открывающейся и закрывающейся фигурными скобками. Увиденный вами код — это заготовка обработчика события, которую автоматически сделал C++Builder. Вам остается только в промежутке между скобками «{» и «}» написать необходимые операторы.

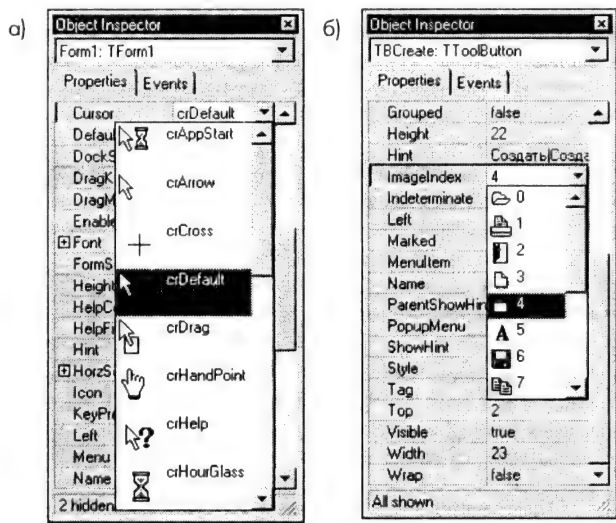
Если вы сделали эти операции, то вернитесь в Инспектор Объектов, выделите в нем, например, событие **OnActivate** и нажмите в нем кнопку выпадающего списка. Вы увидите в нем введенный вами ранее обработчик события **OnCreate** (этот момент изображен на рис. 2.5 б). Если вам надо использовать тот же самый обработчик и в событии **OnActivate**, просто выберите его из списка. Таким образом вы можете избежать дублирования в программе одних и тех же фрагментов кода.

Пользуясь Инспектором Объектов, вы можете получить контекстную справку по свойствам или событиям. Для этого выделите в окне Инспектора Объектов интересующее вас свойство или событие и нажмите клавишу F1.

В C++Builder 5 в Инспектор Объектов введены некоторые дополнительные возможности. Одна из них — отображение в ряде выпадающих списков пиктограмм. На рис. 2.5 а вы уже видели один такой список свойства **Color** с квадратами цветов. Это, конечно, много удобнее, чем абстрактные идентификаторы цветов, которые были в C++Builder 4 и более ранних версиях. На рис. 2.6 показано еще два списка с пиктограммами. Рис. 2.6 а показывает список с изображениями курсоров в свойстве **Cursor**. Из такого списка удобно выбрать вид, который будет приобретать курсор мыши при перемещении над компонентом или формой. А на рис. 2.6 б показан в свойстве **ImageIndex** список пиктограмм компонента **ImageList**, из которых можно выбрать пиктограмму для кнопки проектируемой инструментальной панели или для раздела меню. Этот список особенно удобен, так как его альтернативой в младших версиях C++Builder являлись абстрактные индексы пиктограмм и задавать их было довольно неудобно.

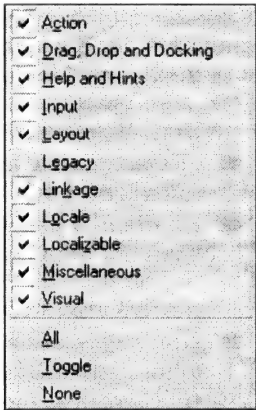
В Инспектор Объектов C++Builder 5 введена также возможность фильтрации свойств и событий и возможность группировать их по категориям. Для того, чтобы воспользоваться этими возможностями, щелкните в окне Инспектора Объектов правой кнопкой мыши. Во всплывшем меню вы можете выбрать раздел View. Вам будет показан ряд категорий свойств (см. рис. 2.7) и событий. Около каждой категории имеется индикатор. Вы можете включить индикаторы только у некоторых категорий и тогда в Инспекторе Объектов вы увидите события и свойства только указанных категорий. Выбор раздела Toggle переключит видимость разделов: те, которые были видимы, станут невидимы и наоборот. Выбор раздела All сделает видимыми все свойства и события, а выбор раздела None сделает все события и свой-

Рис. 2.6
Выпадающие списки Инспектора
Объектов в C++Builder 5 с
пиктограммами



ства невидимыми (правда, непонятно, зачем в этом режиме вообще нужен Инспектор Объектов). Внизу окна Инспектора Объектов указывается, сколько свойств или событий невидимо в данный момент. На рис. 2.5 вы можете видеть, что на странице свойств невидимы (hidden) 2 свойства, а на странице событий видны все события (All shown). Невидимые свойства — следствие выключенного по умолчанию индикатора категории Legacy (см. рис. 2.7).

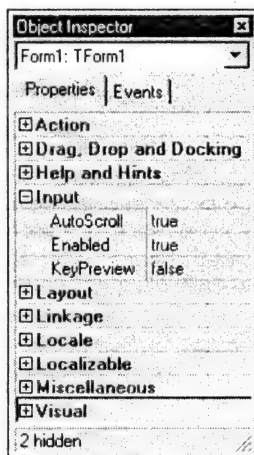
Рис. 2.7
Список категорий свойств



В том же меню, всплывающем при щелчке правой кнопкой мыши в окне Инспектора Объектов, вы можете выбрать раздел Arrange и в нем установить одну из двух возможностей: by Name — упорядочить свойства и события в алфавитной последовательности их имен, или by Category — упорядочить их по категориям. При упорядочивании по категориям форма представления событий и свойств кардинально меняется (рис. 2.8). В окне отображаются категории с символами «+», при щелчке на которых раскрывается список элементов, относящихся к данной категории. При этом некоторые свойства могут попасть одновременно в несколько категорий. Но это не имеет значения: вы можете менять их значения в любой категории и они синхронно изменятся во всех остальных категориях.

Рис. 2.8

Страница свойств Инспектора Объектов, упорядоченная по категориям



Описанные возможности Инспектора Объектов по фильтрации и упорядочиванию информации, введенные в C++Builder 5, существенно упрощают работу с трудно обозримым множеством свойств, присущих многим компонентам библиотеки.

2.2.8 Перетаскивание и встраивание окон в ИСР C++Builder

В Интегрированной Среде Разработки, как и в оконных компонентах C++Builder, широко используется технология Drag&Doc — перетаскивание и встраивание окон. Одно из встраиваемых окон вы уже видели: окно Исследователя Классов — ClassExplorer. По умолчанию оно встроено в окно Редактора Кода (рис. 2.4 б). Есть также еще много встраиваемых окон, которые мы рассмотрим позднее: окно Менеджера Проектов (Project Manager), окно наблюдаемых величин (Watch List) и много других.

Встраиваемое окно можно отличить от обычного по следующим признакам:

- Сокращенная полоса системного меню, включающая обычно только кнопку закрытия окна.
- Наличие в меню, всплывающем при щелчке в окне правой кнопкой мыши, переключателя Dockable — встраиваемое. Если снять метку с этого переключателя, окно перестанет быть встраиваемым. В дальнейшем вы можете опять пометить этот переключатель, и окно снова станет встраиваемым.
- При перетаскивании встраиваемого окна размеры его рамки изменяются, если окно перемещается в пределах другого окна.

Встраивание окон позволяет вам экономить площадь экрана. Для того, чтобы переместить встраиваемое окно, надо потянуть курсором мыши за двойную рамку на одной из его границ (на рис. 2.4 б она на верхней границе окна). При этом можно вынуть его из окна — контейнера и сделать самостоятельным — это так называемое плавающее окно. Для перевода окна в плавающее состояние не обязательно тянуть за двойную рамку — достаточно сделать на ней двойной щелчок. Можно встроить окно ClassExplorer в окно Редактора Кода иначе, чем это принято по умолчанию, например, снизу, чтобы не уменьшать видимую длину строк кода. Во встроеном состоянии можно курсором мыши передвинуть границы окон, практически убрав при желании одно из окон, которое в данный момент не нужно.

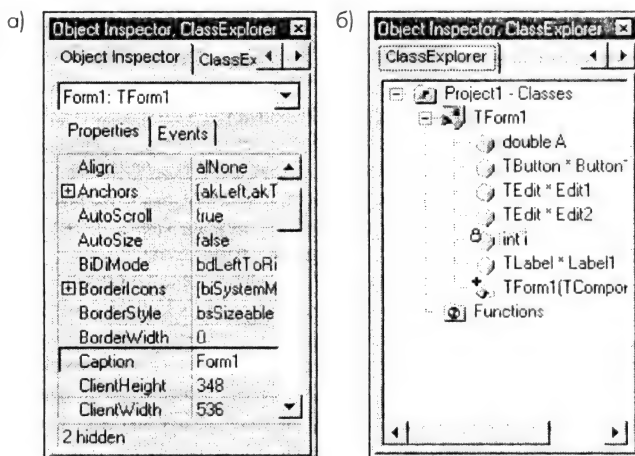
Очень удобно встраивать окна в Инспектор Объектов. Они при этом ложатся на отдельные страницы, совершенно не занимая на экране дополнительного места (рис. 2.9). А когда требуется, вы всегда можете посмотреть требуемое окно, щелк-

нув на его закладке. Поскольку не все закладки могут уместиться в заголовке окна, в нем появляются на уровне закладок кнопки со стрелками, направленными влево и вправо (см. рис. 2.9). С их помощью можно перейти на страницу, закладка которой не видна.

Рис. 2.9

Окно ClassExplorer, встроенное в качестве отдельной страницы в окно Инспектора Объектов:

- а) — открыт Инспектор Объектов,
б) — открыт ClassExplorer



Технология встраивания окон Drag&Doc реализована также в инструментальных панелях. Вглядитесь (рис. 2.1), и вы увидите, что в действительности ИСР содержит 6 панелей, разделенных двойными рамками. Потяните какую-нибудь из панелей (например, палитру компонентов) за эту рамку, и увидите, что вы можете ее перемещать, например, перевести ее в дополнительный третий ряд панелей, чтобы увеличить доступную длину, или вообще перевести в плавающее состояние. Таким образом, вам предоставлены огромные возможности по преобразованию инструментальных панелей. Только будьте осторожны с этими возможностями. Создатели C++Builder неплохо продумали расположение всех органов управления ИСР. И вряд ли стоит что-то кардинально менять в панелях.

Следует также упомянуть, что возможности настройки панелей достаточно широкие. Вы можете делать какие-то из панелей, которые вам сейчас не нужны, невидимыми. Можете настраивать состав отдельных панелей, добавляя и удаляя из них какие-то кнопки. Все процедуры, связанные с настройкой инструментальных панелей, рассмотрены в разделе 14.2.1 главы 14.

2.2.9 Управление конфигурациями окон ИСР

В предыдущих разделах мы рассмотрели несколько окон ИСР: Инспектора Объектов, Редактора Кода, упоминалось окно ClassExplorer. В дальнейшем будет рассмотрено много других окон, облегчающих написание кода и отладку приложения. Каждый пользователь открывает те окна, которые требуются ему для того или иного вида работ, и располагает их удобным для себя образом. Одни окна развернуты, другие свернуты, какие-то окна встроены друг в друга. Так создается удобная для пользователя конфигурация окон. Хотелось бы запоминать эту конфигурацию, чтобы не повторять работу по оборудованию своего рабочего места каждый раз при запуске C++Builder. Запоминание конфигурации можно осуществить несколькими способами.

Можно сделать так, чтобы при очередном запуске C++Builder восстанавливалась конфигурация, которая была на момент завершения предыдущего сеанса работы. Причем не только конфигурация окон, но и загруженный в них проект, с ко-

торым вы работали в последний раз. И даже положение курсора в окне Редактора Коды восстановится тем, которое было в момент окончания предыдущего сеанса. Так что вы сразу можете продолжать работу над тем же проектом с того самого места, на котором остановились. Эта прекрасная возможность реализуется выполнением команд `Tools | Environment Options`. В открывшемся диалоговом окне настроек среды проектирования на странице `Preferences` (см. рис. 14.23 раздела 14.2.10, в котором все это рассмотрено подробнее) в группе опций `Autosave options` надо включить опцию `Project desktop`. Тогда текущая конфигурация и открытые файлы проекта автоматически сохраняются при завершении сеанса работы с `C++Builder` и автоматически восстанавливаются при начале нового сеанса. При включении этой опции конфигурация окон запоминается также в каждом проекте. Так что если вы впоследствии откроете какой-то проект, с которым работали ранее, то восстановится конфигурация всех окон, которые были открыты в момент окончания предыдущего сеанса работы с данным проектом.

Подобную операцию можно выполнить в любой версии `C++Builder`. В `C++Builder 5` введены расширенные возможности сохранения конфигураций. Установив на экране некоторую конфигурацию окон, вы можете выполнить команду `View | Desktops | Save Desktop` или нажать соответствующую ей быструю кнопку (см. таблицу 2.1 в разделе 2.2.3). Появится диалоговое окно, в котором вы должны дать имя сохраняемой конфигурации. Имя может быть записано русским текстом. Эту операцию вы можете повторить несколько раз для разных конфигураций, которые вы используете в своей работе. В результате в выпадающем списке панели выбора конфигурации (см. рис. 2.1 вверху справа) появятся имена составленных вами конфигураций. В дальнейшем, работая с каким-то проектом, вы можете в любой момент выбрать в этом списке одну из конфигураций и она появится на экране. То же самое вы можете сделать командой `View | Desktops` с последующим выбором конфигурации из появившегося списка. При выборе одной из конфигураций файлы, открытые вами в окне Редактора Кода, останутся неизменными. Так что вы продолжите работу с вашим проектом, но уже в новой конфигурации окон.

Уточним, что именно сохраняется в конфигурациях:

- совокупность отрытых окон
- размеры и расположение окон на экране (но не сохраняется их положение в так называемой Z-последовательности, определяющей в случае взаимного перекрытия окон, какие из них находятся поверх других)
- состояние каждого окна (свернутое, развернутое, встроенное в другое окно)
- настройки окна (например, в Инспекторе Объектов сохраняется способ отображения информации — по алфавиту или по категориям и установки фильтрации, во встраиваемых окнах сохраняется состояние индикатора `Dockable`)

Введенные вами конфигурации вы можете впоследствии реорганизовать, удалить ненужные командой `View | Desktops | Delete`.

Конфигурация различных вспомогательных окон, установленная для одной из запомненных конфигураций, используется как конфигурация отладки. Какая именно — устанавливается с помощью команды `View | Desktops | Set Debug Desktop` или соответствующей ей быстрой кнопки (см. таблицу 2.1 в разделе 2.2.3). Пользователю предлагается список сохраненных конфигураций, из которого он должен выбрать конфигурацию отладки. В этом случае именно эта конфигурация будет автоматически загружаться, как только вы запускаете свое приложение на выполнение в режиме отладки.

Пусть, например, вы сделали и сохранили две конфигурации: одну без каких-либо вспомогательных окон сохранили под именем «Умолчание», вторую с окном наблюдения `Watches`, встроенным в окно Инспектора Объектов (см. раздел 2.6.4), сохранили под именем «`Watches`». Выполняя команду установки конфигурации отладки указали в качестве этой конфигурации «`Watches`». Тогда, не-

зависимо от того, какую из конфигураций — «Умолчание» или «Watches» вы указали при работе с проектом, в момент, когда приложение будет запущено на выполнение, установится конфигурация «Watches». После окончания сеанса отладки конфигурация автоматически вернется к исходной.

2.3 Первые шаги — первые собственные приложения

2.3.1 Очень простое приложение

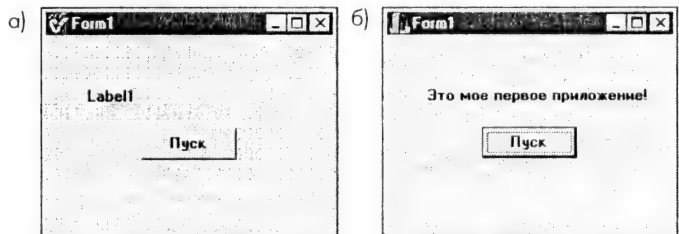
Теперь вы имеете некоторое представление об основных (но далеко не всех) элементах Интегрированной Среды Разработки (ИСР) C++Builder. Самое время попробовать написать первое приложение.

Для начал давайте решим совсем простую задачу: создать приложение, в котором при щелчке пользователя на кнопке появлялась бы какая-нибудь надпись. Выполните для этого последовательно следующие шаги.

1. Запустите C++Builder, если вы еще это не сделали, с помощью меню Windows Пуск | Программы. Если C++Builder уже работает и вы уже делали какие-то эксперименты с формой, то откройте новое приложение. Для этого вам надо выполнить команду File | New Application. Ответьте «No» на вопрос C++Builder, хотите ли вы сохранить изменения в вашем проекте
2. Перенесите на пустую форму, которая открылась вам, кнопку типа **TButton** со страницы Standard палитры компонентов. Для этого выделите пиктограмму кнопки (она шестая слева) и затем щелкните курсором мыши в нужном вам месте формы. На форме появится кнопка, которой C++Builder присвоит имя по умолчанию — **Button1**.
3. Аналогичным образом перенесите на форму с той же страницы Standard палитры компонентов метку **Label** (она на странице третья слева). В этой метке в процессе выполнения приложения будет появляться текст при нажатии пользователем кнопки. C++Builder присвоит метке имя **Label1**.
4. Разместите компоненты на форме примерно так, как показано на рис. 2.10 а. При этом уменьшите до разумных размеров окно формы, так как в вашем первом приложении никаких других компонентов не будет.

Рис. 2.10

Форма (а) и окно в процессе выполнения (б) вашего приложения



5. Выделите на форме компонент **Button1** — кнопку. Перейдите в Инспектор Объектов и измените ее свойство **Caption** (надпись), которое по умолчанию равно **Button1** (имя, которое по умолчанию присвоил этому компоненту C++Builder) на «Пуск».
6. Укажите метке **Label1**, что надписи на ней надо делать жирным шрифтом. Для этого выделите метку, в окне Инспектора Объектов раскройте двойным щелчком свойство **Font** (шрифт), затем также двойным щелчком раскройте подсвойство **Style** (стиль) и установите в **true** свойство **fsBold** (жирный).

7. Сотрите текст в свойстве **Caption** метки **Label1**, чтобы он не высвечивался, пока пользователь не нажмет кнопку приложения.

Теперь вам осталось только написать оператор, который заносил бы в свойство **Caption** метки **Label1** нужный вам текст в нужный момент. Этот момент определяется щелчком пользователя на кнопке. При щелчке в кнопке генерируется событие **OnClick**. Следовательно, обработчик этого события вы и должны написать.

8. Выделите кнопку **Button1** на форме, перейдите в Инспектор Объектов, откройте в нем страницу событий (Events), найдите событие кнопки **OnClick** (оно первое сверху) и сделайте двойной щелчок в окне справа от имени этого события. Это стандартный способ задания обработчиков любых событий. Но перейти в обработчик события **OnClick** (только этого события) можно и иначе: достаточно сделать двойной щелчок на компоненте **Button1** на форме. В обоих случаях вы окажетесь в окне Редактора Кода и увидите там текст:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
}

```

Заголовок этой функции складывается из имени класса вашей формы (**TForm1**), имени компонента (**Button1**) и имени события без префикса **On (Click)**.

9. Если хотите, можете закрыть окно Исследователя Классов, встроенное в окно Редактора Кода (рис. 2.4), так как оно пока вам не нужно и будет только мешать. Закрыть это дополнительное окно можно, щелкнув на кнопке в его правом верхнем углу.

10. Напишите в обработчике оператор задания надписи метки **Label1**. Этот оператор может иметь вид:

```
Label1->Caption = "Это мое первое приложение!";
```

Таким образом, полностью ваш обработчик события должен иметь вид:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Label1->Caption = "Это мое первое приложение!";
}

```

Оператор, который вы написали, означает следующее. Символ «**=**» обозначает операцию присваивания, в которой тому, что написано перед этим символом, присваивается значение того, что написано после символа присваивания. Слева вы написали: **Label1->Caption**. Это значит, что вы присваиваете значение свойству **Caption** компонента **Label1**. Все указания свойств и методов производятся аналогичным образом (см. раздел 1.5.5.2): пишется имя компонента, затем ставятся символы операции стрелка «**->**»: символ минус «**-**» и символ больше «**>**», записанные без пробела. После этих символов пишется имя свойства или метода. В данном случае свойству **Caption** вы присваиваете строку текста «Это мое первое приложение!».

Если вы написали первый идентификатор оператора — **Label1**, поставили символы стрелки и ненадолго задумались, то вам всплывет подсказка, содержащая список всех свойств и методов метки. Это начал работать Знаток Кода, который стремится подсказать вам свойства и методы компонентов, аргументы функций и их типы, конструкции операторов. Вы можете выбрать из списка нужное ключевое слово, нажать клавишу **Enter** и выбранное слово (свойство, метод) окажется вписанным в текст. Можете поступить иначе: начать писать нужное свойство. Тогда Знаток Кода сам найдет по первым введенным символам нужное свойство. Когда вы увидите, что нужное слово найдено, можете его не дописывать, а нажать **Enter**, и Знаток Кода допишет его за вас. Этот инструмент очень удобен во многих случаях, когда вы не очень точно помните последовательность перечисле-

ния параметров какой-нибудь функции, или когда не уверены в имени какого-то свойства (особенно в том, какие буквы этого слова в каком регистре надо писать). Но иногда, когда вы хорошо освоитесь с C++Builder, эти подсказки, может быть, начнут вас раздражать. К тому же, иногда Знаток Кода безо всякого вашего желания дописывает начатое вами слово, причем дописывает не всегда верно. Имейте в виду, что при настройке ИСР вы можете временно отключить Знатка Кода или увеличить задержку, с которой он срабатывает (см. о настройке Знатка Кода в разделе 14.2.6 главы 14).

Итак, ваше приложение готово. Можете откомпилировать и выполнить его. Для этого выполните команду **Run | Run**, или нажмите соответствующую быструю кнопку (см. выше в разделе 2.2.3 таблицу 2.1), или нажмите «горячую» клавишу F9. Если вы ничего не напутали, то после недолгой компиляции, сопровождающейся временным появлением на экране окна компилятора, перед вами появится окно вашего первого приложения. Нажав в нем кнопку «Пуск» вы увидите указанную вами строку текста (рис. 2.10 б). Можете попробовать различные манипуляции с окном: перемещение его, изменение размеров его рамки курсором мыши, свертывание и разворачивание. В заключение закройте приложение, щелкнув на кнопке в его правом верхнем углу.

2.3.2 Немного более сложное приложение

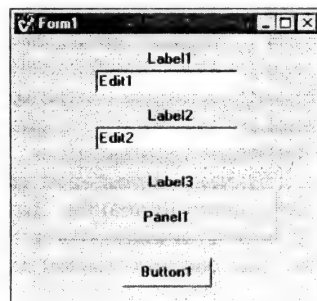
Вас можно поздравить с созданием в C++Builder первого приложения для Windows. А теперь испытайте свои силы в построении чуть-чуть более сложного приложения, которое хоть что-нибудь вычисляет. Создайте приложение, которое при нажатии кнопки перемножало бы два числа, введенных пользователем, и показывало бы результат умножения. Эти числа можете понимать как хотите: как длину двух сторон прямоугольника, и тогда результат — это площадь, или как текущий курс доллара и сумму в долларах — тогда результатом будет рублевый эквивалент суммы и т.п.

При построении этого приложения мы используем новые типы компонентов — окна редактирования **Edit**. Кроме того, для разнообразия, будем выводить результат не в метку **Label**, а в панель **Panel**, просто для того, чтобы испытать новый компонент. При создании этого приложения я уже не буду так подробно описывать каждое действие, а ограничусь только общим описанием интерфейса и необходимых кодов.

Откройте новое приложение. Перенесите на него со страницы библиотеки Standard два окна редактирования типа **TEdit**, одну панель типа **TPanel**, одну кнопку типа **TButton** и три метки типа **TLabel** для надписей. При размещении нескольких компонентов одного типа можно нажать клавишу **Shift** перед щелчком на палитре компонентов, затем указать места на форме, куда надо разместить компоненты, а в заключение нажать кнопку указателя в левой части палитры компонентов, чтобы прервать размещение. Поместите компоненты на форме примерно так, как показано на рис. 2.11.

Рис. 2.11

Форма вашего второго приложения



Измените надписи в метках (свойство **Caption**) на что-то осмысленное. Например, на «Число 1», «Число 2», «Результат» или на «Ширина», «Высота», «Площадь» в зависимости от того, что вы хотите понимать под соответствующими числами. Полезно задать для меток жирный шрифт, как вы это делали в предыдущем примере.

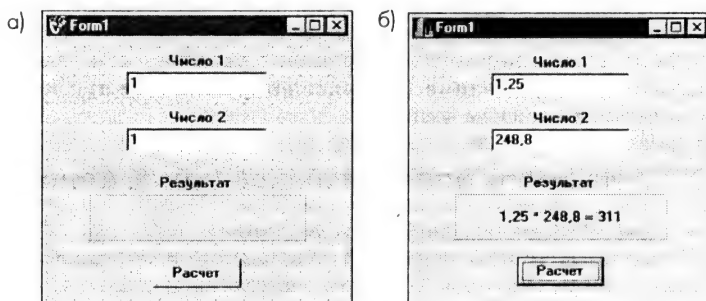
Замените свойство **Caption** вашей кнопки, например, на «Расчет». Очистите свойство **Caption** у панели. В свойстве **Text** (текст) окон редактирования задайте «1» — начальное значение текста.

Попробуйте поварьировать такими свойствами панели, как **BevelInner** и **BevelOuter**, которые определяют вид (утопленный — **bvLowered** или выпуклый **bvRaised**) основного поля и рамки панели. Например, можете установить **BevelInner** = **bvLowered** и **BevelOuter** = **bvRaised**.

В итоге ваша форма приобретет вид, показанный на рис. 2.12 а.

Рис. 2.12

Окончательный вид формы (а) и окна во время выполнения (б)



Осталось написать обработчик щелчка кнопки. Единственный оператор этого обработчика может иметь вид:

```
Panel1->Caption = Edit1->Text + " * " + Edit2->Text + " = "
                + FloatToStr(StrToFloat(Edit1->Text) *
                StrToFloat(Edit2->Text));
```

Попробуем проанализировать этот оператор. Начало его вам уже знакомо: вы присваиваете свойству **Caption** компонента **Panel1** значение выражения, указанного в правой части оператора. Это выражение должно иметь тип строки текста. Начинается строка с текста, введенного пользователем в окно редактирования **Edit1** — этот текст хранится в свойстве **Text**. Затем вы прибавляете к этому тексту символы « * ». Знак «+» в выражениях для строк означает конкатенацию — сцепление двух строк символов. Затем аналогичным образом к строке добавляется текст второго окна редактирования и символы « = ». После этого мы хотим вставить в строку результат перемножения двух целых чисел. Этот результат будет числом и, чтобы вставить его в текст, надо сначала преобразовать это число в строку. Эту операцию выполняет функция **FloatToStr(...)**, которая преобразует заданный ей параметр типа действительного числа в строку символов. Осталось получить само произведение двух чисел. Но числа заданы пользователем в виде текстов — строк символов в окнах редактирования. Прежде, чем перемножать, эти строки надо перевести в числа. Эту операцию выполняют функции **StrToFloat()**, преобразующие символическое изображение числа в его значение типа действительного числа. Знак '*', указанный между двумя функциями **StrToFloat**, обозначает операцию умножения.

Ваше приложение готово. Можете его сохранить — ведь оно все-таки умеет делать нечто полезное. Для этого лучше всего создать отдельный подкаталог (папку Windows) и выполнить команду **File | Save All**. Проще эти команды выполнять с помощью соответствующих быстрых кнопок (см. таблицу 2.1 в разделе 2.2.3).

Откомпилируйте свое приложение и выполните его. Убедитесь, что оно нормально работает (рис. 2.12 б) и мгновенно перемножает самые замысловатые многозначные числа.

Вы создали два простеньких приложения, на которых могли почувствовать процесс написания в C++Builder прикладных программ. Позднее, в последующих разделах вы научитесь создавать действительно сложные и эффективные приложения различного назначения. А пока в дальнейших разделах данной главы давайте рассмотрим методику работы в ИСР C++Builder 5 и соответствующий инструментарий.

2.4 Проекты C++Builder

2.4.1 Организация проекта в C++Builder, основные файлы проектов

Проект C++Builder состоит из форм, модулей с их заголовочными файлами и файлами реализации, установок параметров проекта, ресурсов и т.д. Вся эта информация размещается в файлах. Многие из этих файлов автоматически создаются C++Builder, когда вы строите ваше приложение. Ресурсы, такие, как битовые матрицы, пиктограммы и т.д., находятся в файлах, которые вы получаете из других источников или создаете при помощи многочисленных инструментов и редакторов ресурсов, имеющихся в вашем распоряжении. Кроме того, компилятор также создает файлы. Давайте бегло познакомимся с некоторыми из этих файлов, так как знание того, какие файлы какую информацию содержат, не раз поможет вам в трудных ситуациях.

Когда вы проектируете ваше приложение, C++Builder создает следующие файлы:

Головной файл проекта (.cpp)	C++Builder создает файл .cpp для головной функции WinMain , инициирующей приложение и запускающей его на выполнение
Файл опций проекта (.bpr)	Этот текстовый файл содержит установки опций проекта и указания на то, какие файлы должны компилироваться и компоноваться в проект. Файл сохраняется в формате XML
Файл ресурсов проекта (.res)	Двоичный файл, содержащий ресурсы проекта: пиктограммы, курсоры и т.п. По умолчанию содержит только пиктограмму проекта. Может дополняться с помощью Редактора Изображений (Image Editor)
Файл реализации модуля (.cpp)	Каждой создаваемой вами форме соответствует текстовый файл реализации модуля, используемый для хранения кода. Иногда вы можете сами создавать модули, не связанные с формами
Заголовочный файл модуля (.h)	Каждой создаваемой вами форме соответствует не только файл реализации модуля, но и его заголовочный файл с описанием класса формы. Вы можете и сами создавать необходимые заголовочные файлы

Файл формы (.dfm)	Это двоичный или текстовый файл, который C++Builder создает для хранения информации о ваших формах. Вы можете смотреть этот файл в текстовом виде или в виде формы. Каждому файлу формы соответствует файл модуля (.cpp).
Заголовочный файл компонента (.hpp)	Файл создается при создании вами нового компонента. Вам также часто приходится подключать к проекту эти файлы из библиотеки компонентов C++Builder, расположенные в каталоге Include\VCL
Файл группы проектов (.bpg)	Текстовый файл, создаваемый в C++Builder 5 при создании вами группы проектов
Файлы пакетов (.bpl и .bpk)	Эти двоичные файлы используются C++Builder при работе с пакетами: .bpl — файл самого проекта, .bpk — файл, определяющий компиляцию и компоновку пакета
Файл рабочего стола проекта (.dsk)	В этом текстовом файле C++Builder хранит информацию о последнем сеансе работы с проектом: открытых окнах, их размерах и положении. Благодаря этому файлу в новом сеансе вы сразу видите тот же экран, который был в предыдущем сеансе. Файл создается только если вы включили опцию Autosave options/ Project desktop оболочки (см. раздел 14.2.10)
Файлы резервных копий (.~bp, .~df, .~cp, .~h)	Это соответственно файлы резервных копий для файлов проекта, формы, реализации модуля и заголовочного. Если вы что-то безнадежно испортили в своем проекте, можете соответственно изменить расширения этих файлов и таким образом вернуться к предыдущему не испорченному варианту

Следующая группа файлов создается компилятором:

Исполняемый файл (.exe)	Это исполняемый файл вашего приложения. Он является автономным исполняемым файлом, для которого больше ничего не требуется, если только вы не используете библиотеки, содержащиеся в пакетах, DLL, OCX и т.д.
Объектный файл модуля (.obj)	Это откомпилированный файл модуля (.cpp), который компоуется в окончательный исполняемый файл.
Динамически присоединяемая библиотека (.dll)	Этот файл создается в случае, если вы проектируете свою собственную DLL.
Файл таблицы символов (.tds)	Двоичный файл, используемый отладчиком в процессе отладки приложения.
Файлы выборочной компоновки (.il?)	Файлы с расширением, начинающемся с il (.ilc, .ild, .ilf, .ils), позволяют повторно компоновать только те файлы, которые были изменены после последнего сеанса.

И, наконец, другие файлы Windows, которые могут использоваться C++Builder:

Файлы справки (.hlp)	Это стандартные файлы справки Windows, которые могут быть использованы вашим приложением C++Builder.
Файлы изображений или графические файлы (.wmf, .bmp, .ico)	Эти файлы обычно используются в приложениях Windows для создания привлекательного и дружелюбного пользовательского интерфейса.

Из всех перечисленных файлов (а могут использоваться еще и другие) важными являются файлы **.cpp**, **.h**, **.dfm**, **.bpr**, **.res**. Это те файлы, которые вы, например, должны перенести на другой компьютер, если захотите продолжить на нем работу над своим проектом. Все остальные файлы C++Builder создаст автоматически в процессе компиляции проекта и его отладки.

Главной частью вашего приложения является головной файл **.cpp** с функцией **WinMain**, с которой начинается выполнение вашей программы и которая обеспечивает инициализацию других модулей. Он создается и модифицируется C++Builder автоматически в процессе вашей разработки приложения. Имя, которое вы даете файлу проекта, когда сохраняете его, становится именем исполняемого файла. Сведения об этом файле вы можете найти в разделе 1.5.3.

Все изменения файла проекта при добавлении новых форм, изменении имен форм и т.п. поддерживаются C++Builder автоматически. Если вам необходимо посмотреть исходный файл проекта, надо выполнить команду View | Project Source. Но обычно это вам не требуется.

Предупреждение

Не следует без нужды изменять файл проекта. Это может привести к несогласованности используемых имен и к прочим неприятностям.

Информация о формах C++Builder хранится в трех файлах: **.dfm**, **.cpp** и **.h**. В двоичном или текстовом файле с расширением **.dfm** хранится информация о внешнем виде формы, ее размерах, местоположении на экране и т.д. Щелкнув на форме правой кнопкой мыши, вы можете выбрать из контекстного меню раздел View as Text и увидите в Редакторе Кода файл **.dfm** в текстовом виде. Можете что-то изменить в этом файле. Щелкнув правой кнопкой мыши в окне Редактора Кода на файле **.dfm**, вы можете выбрать в контекстном меню раздел View as Form и опять увидеть файл в виде формы.

Основной файл, с которым вы работаете — файл реализации модуля **.cpp**, в котором хранится код, соответствующий данной форме. В текстовом заголовочном файле с расширением **.h** хранится объявление класса вашей формы. Весь основной текст этого файла C++Builder формирует автоматически по мере проектирования вами формы. Но иногда вам требуется вручную вводить в этот файл объявления каких-то своих функций, типов, переменных. Вы можете загрузить этот файл в Редактор Кода, щелкнув в его окне с файлом реализации модуля **.cpp** правой кнопкой мыши и выбрав из всплывшего меню команду Open Source/Header File.

Имена всех трех файлов, описывающих модуль, одинаковы. Вы задаете это имя, когда в первый раз сохраняете ваш модуль.

Вы можете создавать модули, не привязанные к конкретным формам. Например, в большом приложении полезно иметь модуль (заголовочный файл и файл реализации), содержащий константы, переменные, функции, используемые в различных модулях. Наличие такого модуля позволяет сократить число взаимных ссылок различных модулей. К тому же подобный модуль может использоваться в

разных ваших проектах. Чтобы создать в вашем проекте новый модуль, не связанный с какой-либо формой, надо выполнить команду File | New и в открывшемся окне New Items на странице New щелкнуть на пиктограмме Unit.

Хороший стиль программирования

Полезно создавать в приложении модуль, не связанный с формой, в который помещать описания типов, констант, переменных, функций, используемых другими модулями. Это способствует хорошей структурированности программы, поддерживает единое понимание типов, констант, переменных во всех модулях и уменьшает количество взаимных ссылок модулей друг на друга. Тем самым упрощается модификация и сопровождение программы.

Если вы поочередно работаете над многими проектами, то пространство на диске может неэффективно забиваться ненужными файлами. В этом случае полезно удалять вспомогательные файлы тех проектов, над которыми вы временно не работаете. Прежде всего это относится к файлам `.obj`, `.res`, `.tds`, `.il?`, `.*`. Особо обратите внимание на файлы `.tds`, объем которых может быть очень большим (несколько мегабайт).

2.4.2 Создание и сохранение нового проекта

2.4.2.1 Организация каталогов проекта

Разговор о создании нового проекта начнем с одного важного совета.

Хороший стиль программирования

Заведите себе за правило отводить для каждого нового проекта новый каталог (папку Windows). Удобная структура каталогов существенно облегчает работу над проектами.

Связан этот совет с тем, что если помещать несколько проектов в один каталог, то и вы, и, возможно, C++Builder скоро запутаетесь в том, какие файлы к какому проекту относятся. Размещение проектов в разных каталогах избавит вас в дальнейшем от многих неприятностей.

Если у вас возникает несколько вариантов выполнения проекта, а обычно так и бывает, то желательно внутри каталога проекта создавать подкаталоги для каждого варианта. Удобная структура каталогов существенно облегчает работу над серьезными проектами.

Новый каталог вы можете создать средствами Windows перед началом проекта или, если работаете с Windows 95, 98 или NT, то создавать его можно в диалоговом окне сохранения файлов с помощью соответствующей быстрой кнопки.

2.4.2.2 Создание нового проекта

Начать новый проект можно несколькими способами. Один уже рассматривался выше: команда File | New Application. Можно сделать то же самое с помощью соответствующей быстрой кнопки (см. таблицу 2.1 в разделе 2.2.3). Еще один путь — команда File | New. При выборе этой команды открывается окно New Items (новые элементы), показанное на рис. 2.13.

Это окно является витриной Депозитария объектов (Object Repository) — хранилища образцов компонентов, форм и проектов. Вы можете и сами создавать какие-то свои формы, компоненты, проекты и включать их в Депозитарий. Подробнее о работе с Депозитарием см. в разделе 7.4 главы 7.

В данном случае, мы хотим открыть новый проект. Это можно сделать, щелкнув на пиктограмме Application (приложение), расположенной на странице New (новый). Пиктограмма Application создает уже привычное вам приложение с пустой формой. Вы можете также воспользоваться и некоторыми другими вариантами приложений, пиктограммы которых расположены на странице Projects (рис. 2.14).

Рис. 2.13
Страница New окна Депозитария New Items

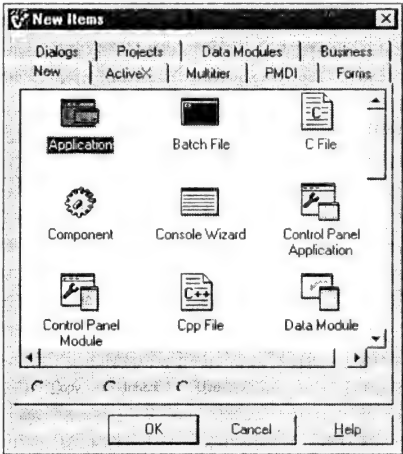
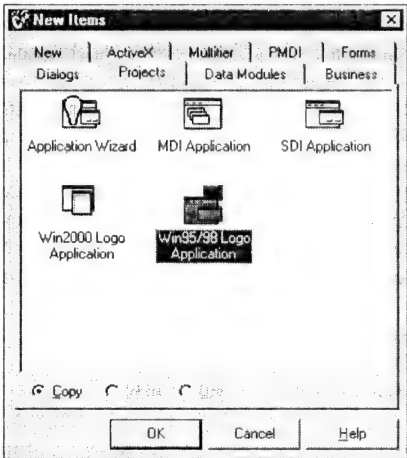
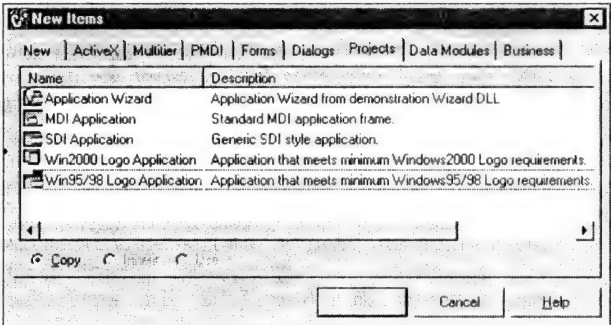


Рис. 2.14
Страница Projects окна Депозитария New Items



Чтобы получить пояснения по содержащимся в Депозитарии объектам, можно щелкнуть в окне правой кнопкой мыши и из всплывшего меню выбрать форму отображения View Details (детали). Форма отображения изменится (рис. 2.15) и вы сможете увидеть в колонке Description краткие пояснения предлагаемых вам вариантов. Например, для приложения Win95/98 Logo Application вы можете прочитать, что это приложение, удовлетворяющее минимальным требованиям логотипа Windows 95/98.

Рис. 2.15
Страница Projects окна Депозитария New Items в режиме просмотра View Details



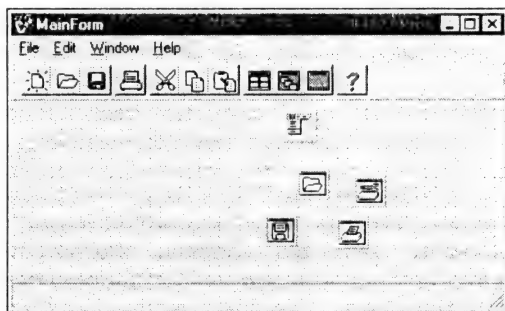
Проекты, выбираемые на этой странице, включают в себя уже не просто пустые формы. Так, щелкнув на выделенной на рис. 2.14 и 2.15 пиктограмме Win95/98 Logo Application, вы создадите приложение почти готового текстового редактора. Приняв его за основу, вы можете, конечно, как угодно его перерабатывать, добавлять новые функции, переводить надписи на русский язык и т.д. Практически такой же редактор, но ориентированный на Windows 2000, получается при выборе пиктограммы Win 2000 Logo Application.

Интересным является проект с пиктограммой MDI Application. Это многооконный текстовый редактор, в котором реализованы все стандартные функции подобных приложений MDI (приложений с множеством документов типа Word, Excel и т.п. — см. о них в разделе 4.5.4 главы 4).

Обратите внимание также на пиктограмму Application Wizard. Выбрав ее, вы попадете под опеку одного из Мастеров (Wizard) C++Builder, который проведет вас через ряд диалоговых окон (подробное описание их вы можете найти в книге [1]), отвечая на вопросы которых вы сможете создать приложение с меню, инструментальной панелью, диалогами и т.д. Пример созданной формы показан на рис. 2.16. В ней имеется заказанная вами в диалоге инструментальная панель, быстрые кнопки которой связаны с соответствующими разделами меню, внесены компоненты-диалоги открытия и сохранения файлов, печати и установки принтера, в быстрых кнопках и в разделах меню внесены тексты ярлычков и развернутых подсказок, которые при работе приложения и при перемещении курсора над разделами меню появляются в строке состояния, помещенной внизу окна. Более того, в текст модуля внесены заготовки кодов с соответствующими комментариями. Остается только перевести заголовки меню и тексты комментариев на русский язык, убрать какие-то не нужные вам разделы меню и добавить нужные, дописать заготовки кодов. Конечно, вам придется в эту заготовку добавлять свои коды. Но значительную часть работы по внешнему оформлению Мастер Приложений сделает за вас.

Рис. 2.16

Форма, сгенерированная Мастером Приложений



Все это, конечно, прекрасно. Возможность пользоваться Мастером Приложений или готовыми проектами из Депозитария выглядит, безусловно, привлекательной. Но я бы не советовал увлекаться этими возможностями. Пользуясь не пустыми эскизами приложений и форм вы всегда рискуете, что в них окажутся какие-то не замеченные вами особенности, которые не позволят приложению работать так, как вам хотелось бы. К тому же приложение будет выглядеть стандартным, лишенным индивидуальности. Когда вы освоитесь с C++Builder, лучше все приложение делать самому, начиная с пустой формы. А проекты из Депозитария, так же как и примеры, поставляемые с C++Builder, полезно посмотреть, изучить, опробовать, перенести какие-то приемы программирования в свои приложения, но не более того.

Можно еще упомянуть о возможности создания с помощью C++Builder консольных проектов — программ, использующих Win32 и запускаемых в Windows 95/98 и NT в окне DOS. Для создания подобного приложения надо вы-

полнить команду File | New и в окне Депозитария на странице New (рис. 2.13) выбрать пиктограмму Console Wizard. Однако, в дальнейшем приложения этого вида мы рассматривать не будем, поскольку они могут найти только очень ограниченное применение.

Из новых Мастеров, появившихся в Депозитарии C++Builder 5 на странице New, можно отметить также Control Panel Application и Control Panel Module, позволяющие создавать приложения, включаемые в «Панель управления» Windows. Пиктограмма Control Panel Application соответствует приложению, размещаемому в «Панели управления». В этом приложении автоматически создается один модуль. Если требуется еще один модуль, он включается пиктограммой Control Panel Module. Приложение представляет собой динамически связываемую библиотеку DLL специального вида, которая дает доступ к конфигурации среды Windows. В модуль приложения автоматически вставляется директива компилятора \$E, обеспечивающая изменение расширения результирующего файла на .cpl.

Объект приложения имеет класс **TAppletApplication**. В этом классе, являющемся контейнером модулей класса **TAppletModule**, определен ряд свойств и прежде всего свойство **AppletIcon** — пиктограмма, которая появляется во время выполнения «Панели управления» в ее окне. Из числа событий прежде всего надо отметить **OnActivate**, наступающее при двойном щелчке пользователя на пиктограмме приложения в окне «Панели Управления».

Установка приложения в «Панели управления» происходит, если вы щелкнете в окне модуля приложения правой кнопкой мыши и выберете во всплывшем меню раздел Install Control Panel Applet. После успешной установки вы можете вызвать из того же окна приложения «Панель управления», выбрав в контекстном меню раздел Launch Control Panel. Раздел того же меню Uninstall Control Panel позволяет удалить ваше приложение из «Панели управления».

2.4.2.3 Сохранение проекта

Первое действие, которое я рекомендовал бы вам выполнить после создания нового проекта — сохранить его.

Хороший стиль программирования

После того, как вы создали приложение с пустой формой, сразу сохраните его в нужном каталоге. И в течение работы над проектом почаще выполняйте сохранение.

Если вы начинаете работу с того, что сохраняете проект, и в дальнейшем регулярно повторяете сохранение, вы можете не опасаться любых неожиданностей типа сбоя компьютера или C++Builder, вызванных техническими причинами или недопустимыми действиями вашего собственного еще не отлаженного приложения при его запуске. Но есть и еще аргумент в пользу немедленного сохранения проекта и модулей. В многооконных приложениях, с которыми вы встретитесь не раз на страницах этой книги, это позволяет сразу задать модулям имена, которые будут использоваться в программе для взаимных ссылок модулей друг на друга. Если вы не выполнили сразу сохранение формы, то вынуждены сначала ссылаться на ее имя по умолчанию, а в дальнейшем изменять эти ссылки.

Хороший стиль программирования

Всегда сохраняйте проект под каким-то осмысленным именем, изменяя тем самым имя проекта, заданное C++Builder по умолчанию. Иначе очень скоро вы запутаетесь в бесконечных программах Project1, лежащих в различных ваших каталогах. К тому же учтите, что имя файла проекта будет в дальнейшем именем вашего выполняемого модуля. Наверное, не очень хорошо будет отдавать заказчику проект с таким странным названием, как Project1.

Сохранить проект можно командой File | Save All. Удобно также использовать соответствующую быструю кнопку. При первом сохранении C++Builder спросит у вас имя файла сохраняемого модуля, а затем — имя файла проекта. Тут надо иметь в виду, что C++Builder не допускает одинаковых имен модулей и проектов. Это естественно, поскольку файлы модуля и головной файл проекта имеют одинаковое расширение .cpp и, значит, должны различаться именами, чтобы не затереть на диске друг друга.

Предупреждение

Не задавайте одинаковые имена различным файлам и вообще стремитесь не использовать в проекте повторяющиеся имена. C++Builder этого не приемлет.

Хороший стиль программирования

Всегда при сохранении модулей задавайте осмысленные имена файлов.

Почему не стоит соглашаться при сохранении с именами, предлагаемыми C++Builder по умолчанию — Unit1, Unit2 и т.п.? Имя файла модуля будет и именем самого модуля, и именем его заголовочного файла. В проекте с одним модулем имя Unit1, возможно, не так уж и страшно. Но если у вас в проекте будет несколько модулей, то вряд ли вам будет удобно все время помнить, что такое Unit1, а что такое Unit5. Кроме того представьте себе, что вы на некоторое время прекратили работу над своими проектами, а потом опять ее возобновили. Или сопровождение ваших проектов осуществляется не вами, а кем-то другим. Вы уверены, что легко будет разобраться в многочисленных Project1 и Unit1 непонятного назначения?

Хороший стиль программирования

При разработке простых прототипов проектов удобно задавать имена файлов проектов и модулей одинаковыми, различая их только префиксом «Р» или «U» соответственно. Например, файл проекта — PEditor1, файл модуля — UEditor1. Это упрощает понимание того, какие файлы к какому варианту проекта относятся, поскольку на начальных стадиях проектирования у вас может появиться несколько альтернативных вариантов одного и того же проекта.

Итак, рекомендации по созданию нового проекта сводятся к следующему.

- Создайте новый каталог для своего нового проекта.
- Создайте новый проект командой File | New Application.
- Сразу сохраните проект и файл модуля командой File | Save All.

В последующих сеансах работы вы можете открыть сохраненный проект командой File | Open Project. Но если вы работали с проектом недавно, то много удобнее открыть его командой File | Reopen. А еще удобнее воспользоваться стрелочкой рядом с соответствующей этой команде быстрой кнопкой (см. раздел 2.2.3). В обоих случаях открывается окно, в котором вы легко найдете несколько проектов, с которыми работали в последнее время, а также ряд файлов, с которыми вы работали.

Имеется еще более удобный способ автоматически открывать при загрузке C++Builder тот проект, с которым вы работали в предыдущем сеансе. Для того, чтобы обеспечить себе такую возможность, надо выполнить команду Tools | Environment Options. В открывшемся многостраничном диалоговом окне надо перейти на страницу Preferences и включить индикатор Project desktop группы опций автосохранения Autosave options (подробности см. в разделе 14.2.10). Тогда при каждом очередном запуске C++Builder будет загружаться ваше последнее приложение предыдущего сеанса и будут открываться все окна, которые были открыты в момент предыдущего выхода из C++Builder. Это очень удобно, если вы намерены продолжать работу над тем же проектом.

Вы можете сохранить свой проект в Депозитории. Впоследствии при появлении схожей задачи вы можете заимствовать его оттуда, что-то в нем изменить и таким образом сэкономить себе много времени. Методика включения своих проектов и форм в Депозиторий подробно рассмотрена в главе 7 в разделе 7.4.

2.4.3 Менеджер проектов

Менеджер Проектов (Project Manager) — это инструмент управления группами проектов. Группа проектов — это термин обозначающий группу родственных проектов, с которыми удобно работать параллельно. Например, это могут быть проекты клиента и сервера, обменивающихся какой-то информацией, или проект, создающий библиотеку DLL, и проект, использующий ее.

Открывается Менеджер Проектов командой View | Project Manager. Если в данный момент нет открытого проекта, то в окне Менеджера Проектов будет написано <No Project Group>. Если же какой-то проект открыт, то вид окна Менеджера Проектов показан на рис. 2.17 а. В данном случае формируется группа с именем по умолчанию **ProjectGroup1**, содержащая только один открытый проект. В окне виден состав проекта. Около строк, относящихся к различным модулям, вы можете видеть символ плюс «+». Сделав на нем щелчок, можно раскрыть состав этого модуля. Сделав двойной щелчок на строке, относящейся к тому или иному модулю или форме, вы увидите соответственно или текст модуля в окне Редактора Кода, или указанную форму. Таким образом, Менеджер Проектов дает удобный способ навигации по модулям сложного проекта.

Рис. 2.17
Окно Менеджера Проектов с одним (а) и двумя (б) проектами в группе



Кнопка New (новый) позволяет добавить в группу новый проект. Действие кнопки Remove (удалить) зависит от того, какая строка в Менеджере Проектов в этот момент выделена. Если выделен модуль или форма, то удаляться будет именно этот объект. А если выделена строка файла .exe, то удаляться будет сам проект. Впрочем, в любом случае предварительно появится диалоговое окно с запросом, действительно ли вы хотите удалить тот или иной объект.

Вы можете также воспользоваться в работе всплывающим меню, которое вызывается при щелчке правой кнопкой мыши. Если в этот момент у вас будет выделен в окне Менеджера Проектов интересующий вас модуль, то во всплывшем меню вы увидите разделы Open (открыть), Remove From Project (удалить из проекта), Save (сохранить), Save As (сохранить как) и ряд других. Если вы выделили в окне не модуль, а проект — имя файла .exe, то всплывет уже другое меню, в котором будут, в частности, разделы Add (добавить новую форму или модуль), Remove File (удалить файл — появится диалоговое окно, в котором вы можете выбрать удаляемый модуль), Save (сохранить проект) и ряд других.

Добавить в группу еще один проект можно двумя способами.

Если вы хотите добавить новый проект, вы можете щелкнуть в окне Менеджера Проектов на кнопке New или выделить в окне имя группы проектов, щелкнуть правой кнопкой мыши и выбрать из всплывшего меню раздел Add New Project (добавить новый проект). Вы попадете в уже рассмотренное нами окно Депозитария и можете выбрать в нем вид открываемого проекта, в частности, Application — новый проект с пустой формой.

Если же вы хотите добавить в группу один из разработанных вами ранее проектов, то, выделив в окне имя группы проектов и щелкнув правой кнопкой мыши, надо выбрать из всплывшего меню раздел Add Existing Project (добавить существующий проект). Далее в обычном окне открытия файла вы можете указать добавляемый файл проекта.

При добавлении в группу нового проекта в окне Менеджера Проектов будет отображаться уже информация о всех проектах группы (рис. 2.17 б).

Вы можете сохранить файл вашей группы, если выполните команду File | Save As или если выделите в окне Менеджера Проектов имя группы, щелкнете правой кнопкой мыши и из всплывшего меню выберете команду Save Project Group или Save Project Group As. Группа проектов сохраняется в текстовом файле с расширением .bpg. В дальнейших сеансах работы вы можете открыть этот файл той же командой File | Open Project или File | Reopen, которой вы открываете проект.

Окно Менеджера Проектов — встраиваемое, так что вы можете встроить его, например, в окно Инспектора Объектов и оно не будет занимать на экране лишнего места.

Теперь посмотрим, что дает нам объединение нескольких проектов в группу. Прежде всего, вы можете параллельно работать над всеми проектами группы, открыв их модули в окне Редактора Кода. Для того, чтобы открыть нужный модуль, достаточно сделать двойной щелчок на его имени в окне Менеджера Проектов. Вы можете также выполнить любой проект группы. В каждый данный момент активным является один из проектов. Имя активного проекта выделено в окне Менеджера Проектов жирным шрифтом. Оно же видно в выпадающем списке в верхней части окна Менеджера Проектов. Например, в окне рис. 2.17 б активным является проект Pmess2. Если вы выполните команду Run | Run или нажмете F9, то именно этот проект будет компилироваться и выполняться. Вы можете сделать активным другой проект. Это можно сделать, выделив нужный проект в окне Менеджера Проектов и щелкнув на кнопке Activate. Второй способ — выбрать в выпадающем списке проектов имя соответствующего проекта и этот проект будет активизирован. Третий способ — щелкнуть на имени проекта, который надо активизировать, правой кнопкой мыши и выбрать во всплывшем меню команду Activate. Четвертый способ добиться того же — нажать кнопку справа от быстрой кнопки выполне-

ния в панели быстрых кнопок ИСР и из выпавшего списка выбрать тот выполняемый файл, который вы хотите выполнять.

Описанные приемы позволяют, если вы работаете в Windows NT, одновременно выполнять и отлаживать несколько проектов одной группы. Для этого сначала надо выполнить компиляцию всех проектов командой Project | Build All Projects. После этого вы можете активизировать один файл и запустить его на выполнение. Затем, не закрывая его, вернуться в ИСР C++Builder, активизировать другой проект группы и тоже запустить его на выполнение. При этом у вас окажется два выполняемых файла проекта, причем вы можете отлаживать их совместную работу (о способах отладки программ будет рассказано позднее).

В Windows 95/98 такая параллельная отладка нескольких проектов невозможна. При необходимости выполнять одновременно несколько проектов вы можете один из них запустить на выполнение в режиме отладки из C++Builder, а остальные надо запустить вне C++Builder средствами Windows, например, программой «Проводник».

Отметим еще одну новую особенность Менеджера Проектов, введенную в C++Builder 5 — возможность задания локальных опций для какого-то узла дерева файлов. Это позволяет компилировать проект с разными опциями для разных модулей. Например, для уже хорошо отлаженных модулей вы можете убрать включение в выполняемый файл отладочной информации, убрать отображение сообщений компилятора и т.п. Это сократит размер выполняемого модуля и ускорит компиляцию.

Настройка компилятора подробно рассмотрена в разделе 14.2.9. А пока ограничимся изложением техники задания именно локальных опций. Читатель может пока пропустить этот материал и вернуться к нему после того, как ознакомится с опциями настройки.

Чтобы задать локальные опции компиляции, надо выделить в окне Менеджера Проектов узел файла **.cpp** или **.c** и выбрать в контекстном меню, всплывающем при щелчке правой кнопкой мыши, раздел Edit Local Options. Вы попадете в окно опций проекта (см. раздел 14.2.9), в котором будут видны только те страницы, на которых можно задавать локальные опции. При изменении каких-то опций они будут выделяться цветом, что позволит вам в дальнейшем легко определить, что именно вы изменяли. Отличие окна опций проекта при задании локальных опций от обычного вида этого окна (см. например, рис. 14.15) будет заключаться в том, что внизу будет отображаться сообщение «Right Click to Revert». Его смысл в том, что щелкнув правой кнопкой мыши вы можете повлиять на вводимые вами локальные опции. При таком щелчке всплывет контекстное меню. Его раздел Revert All позволяет отказаться от всех введенных ранее локальных изменений. В этом случае для данного узла начнут применяться опции, установленные для всего проекта. Раздел контекстного меню Change Override Color позволяет изменить цвет выделения локальных опций.

Если для какого-то модуля вы ввели локальные опции, узел этого модуля в окне Менеджера Проектов будет отмечен галочкой. Это будет служить вам напоминанием о том, что данный модуль компилируется в необычном режиме.

2.4.4 Управление проектами

2.4.4.1 План работ — список To-Do List

Список To-Do List, появившийся только в C++Builder 5, представляет собой запись тех текущих задач, которые надо решить при разработке проекта. Эти задачи могут заноситься в список непосредственно или передаваться в него из кода ваших модулей. Ведение списка облегчает планирование работ, позволяет ранжировать задачи, отслеживать очередность и своевременность их решения.

Список может быть создан для каждого проекта и хранится в файле с расширением **.todo**. Посмотреть список To-Do List, связанный с текущим проектом, можно командой View | To-Do List. Если вы выполните эту команду, перед вами откроется окно, вид которого представлен на рис. 2.18.

Рис. 2.18

Окно списка To-Do List

To Do Items				
Action Item	Module	Owner	Category	
<input type="checkbox"/> Главная форма	1	Иванов	Прототип	
<input checked="" type="checkbox"/> Форма документа	1	Петров	Архитектура	
<input type="checkbox"/> Главное меню	2 ... \Urichedit.cpp	Сидоров	Код	
<input type="checkbox"/> Главная форма	5	Иванов	Полный вариант	
<input type="checkbox"/> Форма документа	5	Петров	Полный вариант	
5 items (0 hidden)		4 items pending		

Список содержит следующие столбцы:

Action Item	Задачи, подлежащие решению. Для каждой задачи отображается:	
Индикатор	Указывает, решена ли уже эта задача. Флажок индикатора означает, что решена. При этом, как видно из рис. 2.18 (вторая строчка) строка задачи отображается зачеркнутой. Если список отображается с выключенной опцией Show Completed Items («Показывать решенные задачи» — об опциях будет рассказано позднее), то выполненные задачи вообще не появляются в списке	
Пиктограмма	Указывает характер задачи: пиктограмма окна соответствует задаче, внесенной непосредственно в список, а пиктограмма модуля (на рис. 2.18 в третьей строчке) соответствует задаче, внесенной в код модуля. Если пиктограмма модуля выглядит серой, это означает, что данный модуль не является частью текущего проекта	
Текст	Описывает задачу. Если текст отображен серым, это означает, что задача внесена в код модуля, являющегося частью данного проекта, но не открытого в данный момент в Редакторе Кода. Двойной щелчок на такой задаче приведет к открытию соответствующего модуля в Редакторе Кода	
Priority	В заголовке столбца указывается восклицательный знак. Указывает уровень приоритета задачи. Наивысшему уровню приоритета соответствует 1, самый низкий уровень — 5. Уровень 0 указывает на отсутствие приоритета у данной задачи	
Module	Содержит имя модуля, в котором введена задача. Этот столбец списка заполняется автоматически	
Owner	Указывает ответственного за решение данной задачи	
Category	Указывает класс данной задачи	

Щелчок правой кнопкой мыши на списке вызывает контекстное меню, в котором вы можете выбрать ряд команд и опций:

Add	Добавить новую задачу в список. В диалоговом окне вы можете указать ее текст (Text), класс(Category), приоритет (Priority) и ответственного (Owner)
Delete	Удалить задачу из списка. То же самое вы можете сделать проще, выделив задачу и нажав кнопку Delete
Edit	Редактировать описание задачи. Вы попадаете в диалоговое окно, аналогичное окну ввода новой задачи, в котором можете отредактировать необходимую информацию. Можете также отметить завершение выполнения задачи, установив индикатор Done. Впрочем, выполнение вы можете отметить и непосредственно в окне списка
Sort	Сортировка отображения списка: Action Item — по алфавитной последовательности задач, Status — сначала отображаются решенные задачи, затем нерешенные, Type — сначала в алфавитном порядке располагаются задачи, введенные непосредственно в список, затем введенные в модули, Priority — по приоритету, Module — в алфавитной последовательности модулей, Owner — в алфавитной последовательности ответственных, Category — в алфавитной последовательности классов задач. Сортировку можно осуществить и проще, щелкнув в окне списка на заголовке соответствующего столбца
Filter	Фильтрация задач, отображаемых в списке. Вы можете указать классы задач (Categories), ответственных (Owners) и типы задач (Item types), которые хотите отображать. При выборе типов задач вы можете установить Current project source files — задачи, добавленные в файл проекта, Open source files — задачи, добавленные в открытые файлы модулей, Project To-Do file — задачи, добавленные непосредственно в список
Show Completed Items	Опция, включающая отображение уже выполненных задач
Show ToolTips When Clipped	Опция, включающая при перемещениях курсора мыши отображение во всплывающих окнах полного текста, если он не виден полностью в колонке
Copy As	Копирование списка в виде текста (Text) или в виде таблицы HTML (HTML Table)
Table Properties	Задание атрибутов отображения таблицы (в частности, русских заголовков) при использовании опции Copy As HTML Table

Рассмотренная выше команда контекстного меню Add позволяет добавлять задачи непосредственно в список. Добавление задачи в файл модуля или проекта осуществляется из окна Редактора Кода. Откройте в нем требуемый файл и в нужном вам месте кода щелкните правой кнопкой мыши и выберите из всплывшего меню команду Add To-Do Item. Вы попадете в то же диалоговое окно, что и при выполнении команды Add и сможете в нем указать все атрибуты задачи. В результате задача не только включится в список, но оператор, соответствующий ей, включится в код. Впрочем, вы можете не использовать меню, а просто записать в коде соответствующий оператор. Синтаксис этого оператора следующий:

```
/* TODO|DONE [n] [-o<ответственный>] [-c<класс>] : <текст> */
```

Весь оператор заключается в скобки комментария (`/* */`). Можно вместо этих скобок записывать перед оператором два слеша `/**/`. Регистр при записи оператора не учитывается. Ключевое слово **TODO** используется для незавершенных задач. Если задача выполнена, это слово заменяется на **DONE**. Обязательными элементами оператора являются только ключевое слово и текст. Все остальные опции могут отсутствовать.

Значение опции **n** указывает приоритет задачи. Опция **-o** указывает ответственного за решение данной задачи. Опция **-с** устанавливает класс задачи.

Например, оператор

```
/* TODO 2 -oСидоров -сКод : Главное меню */
```

задает задачу с именем «Главное меню», класса «Код» с приоритетом 2, за которую отвечает Сидоров.

Введенные в текст операторы отображаются в окне списка и автоматически изменяются при работе со списком (например, при включении индикатора выполнения ключевое слово **TODO** автоматически заменяется на **DONE**).

Исследователь Классов ClassExplorer (см. раздел 2.5.3.2), который в C++Builder 5 существенно облегчает ввод в класс свойств и методов, автоматически включает при этом в текст операторы списка To-Do List в те места кода, в которых вам требуется ввести собственный текст. Это очень поможет вам, напоминая об еще не завершенных фрагментах кода.

2.4.4.2 Система управления большими проектами Borland TeamSource

Система Borland TeamSource позволяет организовать управление большими проектами, разрабатываемыми рядом исполнителей или несколькими группами исполнителей. Впрочем, даже при индивидуальной разработке проекта после того, как вы создали в C++Builder несколько приложений или вариантов приложений, система TeamSource способна оказать большую помощь в организации ваших программ. Вы можете захотеть заархивировать прежние версии приложения прежде, чем начинать работу над новой версией, чтобы потом иметь возможность вернуться к прежней версии. Ну а если проект большой и над ним работает группа программистов, то требуется координация их работы, необходимо обеспечить доступ программистов к тем или иным общим для них файлам и в то же время необходимы гарантии, что один программист случайно не уничтожит или не изменит файлы, над которыми работает другой программист.

Система TeamSource позволяет решать указанные задачи, возникающие при разработке крупных проектов или большой серии мелких проектов. Система позволяет архивировать файлы, управлять доступом к файлам, блокируя их непредусмотренные изменения, позволяет избежать путаницы при работе с множеством версий проекта.

Система позволяет сохранять в архиве все удачные версии каждого файла. Программист может извлечь рабочие файлы из архива, что-то изменить в них, отладить и затем отправить назад в архив, отметив сделанные изменения, дату и автора этих изменений. Возможность блокировки обмена файлами с архивом предотвращает несанкционированное изменение файлов. Вы можете также для каждого файла отслеживать все произведенные в нем изменения на каждом этапе разработки: кто, когда и зачем делал те или иные изменения.

Поскольку система Borland TeamSource включена не во все версии C++Builder 5, мы ограничимся только ее общим описанием. А детали работы с этой системой вы можете найти в книге [1].

Основное понятие, используемое в работе с TeamSource — проект TeamSource. Это файл с расширением `.crj`. Он содержит сведения о включенных в проект TeamSource файлах проекта C++Builder, о пользователях, допущенных к ним, и о вариантах доступа (только чтение, чтение и запись, административный доступ). В

проект TeamSource передаются сведения о *локальных каталогах*, в которых расположены управляемые файлы проекта C++Builder.

Файлы проекта TeamSource размещаются обычно в подкаталогах некоторого каталога, который мы назовем головным. Один из подкаталогов — *Archives* предназначен для хранения архивированных копий версий контролируемых файлов проекта C++Builder. В этом же подкаталоге располагается файл проекта *.cpj*. Архив может иметь структуру дерева, отображающего соотношения между управляемыми файлами. Файлы в архиве читать нельзя. Поэтому проект TeamSource может содержать еще один подкаталог — *Source*, который зеркально отображает структуру архива, но файлы которого можно читать, а при желании — и редактировать.

Еще один подкаталог проекта TeamSource — *History* содержит информацию об истории проекта, о том, когда, кем и зачем создавались различные версии. И, наконец, подкаталог *Locks* содержит информацию о блокировке проекта различными пользователями. Такая блокировка, в частности, необходима, когда в структуре проекта TeamSource производятся какие-то изменения.

Система TeamSource включает работу с двумя вариантами управления версиями: PVCS Version Manager и ZLib. Вариант ZLib (Borland.zlib включен в TeamSource) обеспечивает более высокую степень сжатия нетекстовых архивных файлов, но зато лишен удобных возможностей автоматического объединения изменений, присущих PVCS. Для использования обоих вариантов управления версиями в каталоге, в котором расположен файл *TeamSrc.exe*, должны находиться файлы *IZLib.tsx* и *IPVCS.tsx*. Если при установке системы TeamSource обнаруживается, что на компьютере не установлена PVCS, то второй из указанных файлов не создается.

TeamSource является самостоятельным программным продуктом, который может выполняться из своего каталога (файл *TeamSrc.exe*, папка ...\\Borland TeamSource, пиктограмма TeamSource). Вызов TeamSource может также осуществляться из ИСП C++Builder 5 командой главного меню Tools | TeamSource (если такой команды в меню нет, значит в вашу версию C++Builder система TeamSource не включена).

К сожалению, в рамках данной книги нет возможности рассматривать методику работы с TeamSource. Те, кто заинтересовался этим инструментом, могут получить сведения о работе с TeamSource в книге [1].

2.5 Основные проектные операции при создании приложения

2.5.1 Включение в проект новой формы

2.5.1.1 Зачем надо включать новые формы

Во многих случаях ваш проект будет содержать не одну, а несколько форм. Кроме того, вы, может быть, захотите убрать из нового проекта пустую форму и включить вместо нее другую, разработанную ранее вами или кем-то другим. Например, если вы для какого-то своего проекта разработали форму, запрашивающую пароль пользователя, или форму с информацией о программе и вашим красивым логотипом, то не имеет смысла в новом проекте создавать их заново. Форму с паролем вы могли бы взять из прошлого приложения вообще без каких-либо изменений, а в форме информации о программе вам достаточно сменить только имя программы.

Хороший стиль программирования

Для всех включаемых в проект форм, даже если в вашем проекте всего одна форма, задавайте уникальные имена (свойство Name), изменяя установленные C++Builder по умолчанию. Это существенно облегчит вам повторное использование ваших форм, так как позволит избежать дублирования имен при включении прежней формы в новый проект.

2.5.1.2 Включение в проект новой формы

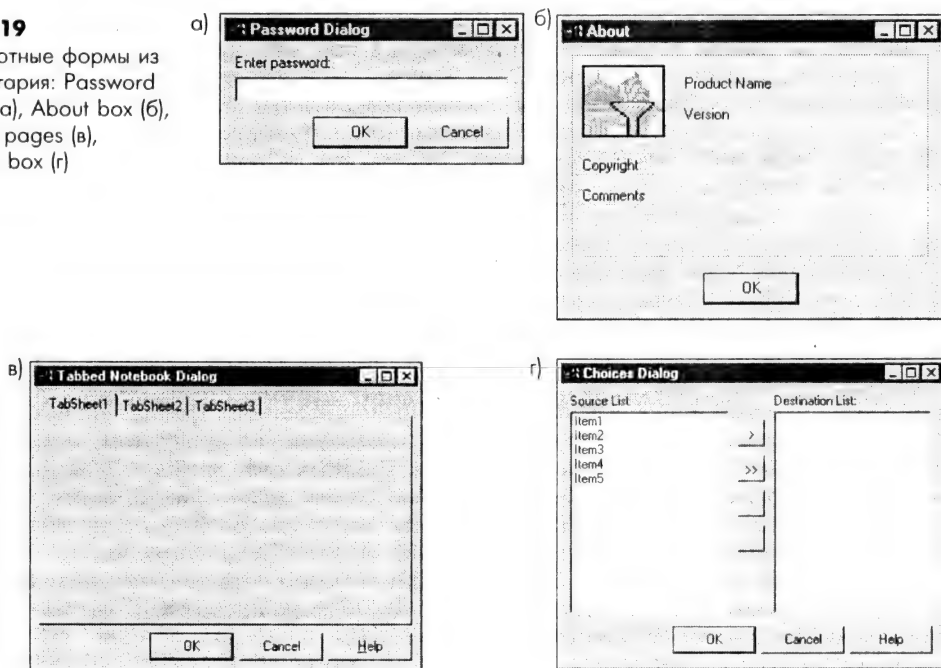
Включение в проект новой формы может производиться различными способами.

Если вы хотите включить новую пустую форму, вам достаточно выполнить команду `File | New Form` или нажать соответствующую быструю кнопку.

Вы можете также воспользоваться различными формами, хранящимися в Депозитарии на страницах `Forms` и `Dialogs`. Примеры этих форм приведены на рис. 2.19.

Рис. 2.19

Стандартные формы из Депозитария: Password Dialog (а), About box (б), Tabbed pages (в), Dual list box (г)



Некоторые из готовых форм очень просты, фактически не содержат выполняемых кодов и, на мой взгляд, много проще создать их самому. Другие включают в себя достаточно сложные коды и ими имеет смысл воспользоваться. Несмотря на это, я бы рекомендовал все-таки использовать всегда пустую форму. Применение шаблонов лишит ваше приложение индивидуальности и единства стиля различных окон. А это очень важный критерий хорошего приложения. Заготовками, имеющимися в C++Builder, лучше воспользоваться для их изучения, для заимствования каких-то фрагментов кода или принципов организации пространства окна, но не использовать их целиком. Впрочем, если ваша задача заключается в быстром создании прототипа приложения, можно воспользоваться и готовыми формами, заменив их в окончательном варианте на свои собственные.

2.5.1.3 Совместное владение формой несколькими приложениями

Теперь рассмотрим случай, когда вам требуется включить в свой проект форму, разработанную ранее вами или кем-то еще для другого проекта. Тут возможны несколько вариантов действий, имеющих различные последствия.

Можно включить готовую форму в проект командой `Project | Add to Project` или соответствующей быстрой кнопкой. При этом, если включаемая форма имеет то же имя, которое имеет одна из уже имеющихся в проекте форм (например, **Form1**, если вы не привыкли изменять имена форм, принимаемые C++Builder по умолча-

нию), то вы получите предупреждение вида: «The project already contains a form or module named Form1» — «Проект уже содержит форму или модуль с именем Form1». В результате форма в проект не включится. Аналогичный вариант будет, если вы не следуете в своей работе уже дававшемуся совету присваивать модулям уникальные имена. Если в проекте уже имеется модуль **Unit1** и вы пытаетесь включить из другого каталога модуль формы, тоже имеющий имя **Unit1**, то вам будет выдано такое же предупреждение и новый модуль в проект не включится.

Разрешить подобные конфликты можно следующим образом. Переименуйте в вашем проекте форму, вызвавшую конфликт (задайте для нее новое имя в свойстве **Name**). Если конфликт вызван совпадением имен модулей, то сохраните конфликтующий модуль командой **File | Save As**, дав ему новое имя. После этого можете повторить попытку добавления в проект новой формы.

Предупреждение

Учтите, что форма, содержащаяся в одном приложении и включенная описанным способом в другое приложение, становится общей для обоих приложений. Если вы сделаете в ней какие-то изменения, а потом перекомпилируете оба приложения, то внесенные изменения отразятся на обоих приложениях.

Введенное описанными действиями совместное владение несколькими приложениями одной и той же формой имеет свои плюсы и минусы. Если эти приложения представляют собой некую группу связанных друг с другом приложений, рассчитанных на применение одними и теми же пользователями, то наличие общих форм можно только приветствовать. Какие-то усовершенствования, введенные в подобной форме, согласованно отобразятся во всех использующих ее приложениях (после их перекомпиляции). Например, если это форма, запрашивающая пароль пользователя, то, конечно, хорошо, если она будет общей (и значит идентичной) в различных приложениях.

Если же приложения, совместно использующие форму, совершенно разные или форма используется в них для разных целей, то, введя изменения в форму в одном приложении, вы рискуете испортить прежнее приложение, если соберетесь его перекомпилировать. Например, изменив имя программы в форме, описывающей новое приложение, вы невольно введете то же имя и в прежнее приложение, то будет, несомненно, ошибкой.

2.5.1.4 Создание отдельной копии формы

Чтобы избежать совместного владения формой несколькими приложениями, после того, как вы включили в новое приложение форму из другого приложения, перейдите в окне Редактора Кода в модуль этой формы и выполните команду **File | Save As**, сохранив модуль в каталоге нового приложения и, если хотите, под другим именем (имя изменять не обязательно). В этом случае разные приложения будут использовать совершенно разные копии одной формы и изменения одной из них не затронут другие приложения.

Можно, конечно, создать копию формы и другими способами. Во-первых, вы можете создать ее средствами Windows или MS DOS, просто скопировав соответствующие файлы из одного каталога в другой. Только не забудьте при этом, что форма — это не только файл модуля **.cpp**, но еще его заголовочный файл **.h** и файл изображения **.dfm**. Так что копировать надо все три файла. Файл объектного модуля можно не копировать, так как он будет создан C++Builder в процессе компиляции.

Еще один способ создания автономной копии формы — использование меню C++Builder. Вы можете в любой момент последовательно выполнить команды **File | Open**, указав файл открываемой формы, и команду **File | Save As**. Первая из этих команд откроет форму, а вторая сохранит ее в указанном вами каталоге под указанным именем. Преимущество такой операции заключается в том, что вам не нужно

думать о совокупности сохраняемых файлов. C++Builder автоматически скопирует не только файл `.cpp`, но и файлы `.h` и `.dfm`.

Если вы заимствуете формы из Депозитария командой `File | New` (см. рис. 2.19 с примерами этих форм), то имеется несколько вариантов такого заимствования: копирование, наследование, использование. Вопросы заимствования форм из Депозитария рассмотрены в главе 7 в разделе 7.4. Там же рассказано, как можно сохранять в Депозитарии спроектированные вами формы и как создавать иерархию форм.

2.5.1.5 Просмотр форм и модулей без включения их в проект

Нередко у вас может возникнуть потребность в процессе работы над проектом посмотреть формы и модули из других проектов. Это могут быть формы примеров, поставляемых с C++Builder. Или могут быть ранее разработанные вами формы, из которых вы хотите взять какие-то операторы, решающие задачу, близкую к той, которой вы заняты в данный момент.

Открыть некоторый существующий модуль, не включая его в текущий проект, очень легко. Достаточно выполнить команду `File | Open`, и указанный вами файл модуля окажется в окне Редактора Кода. Вы можете просматривать его и соответствующую ему форму, копировать через буфер обмена Clipboard какие-то операторы или компоненты в свои модули.

Когда необходимость в открытом модуле отпадет, щелкните на его коде правой кнопкой мыши и из всплывшего меню выберите команду `Close Page`. Страница Редактора Кода с текстом данного модуля и его форма будут закрыты.

2.5.2 Размещение компонентов на форме

2.5.2.1 Перенос компонентов со страниц библиотеки на форму

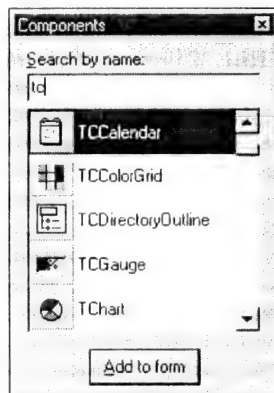
Для того, чтобы перенести на форму компонент, надо открыть соответствующую страницу палитры компонентов и найти на ней нужный компонент. В поиске вам очень помогут ярлычки, появляющиеся, если вы задерживаете курсор над той или иной пиктограммой. Выделите курсором нужный вам компонент, а затем щелкните мышью в том месте формы, куда вы хотите поместить компонент. Возможен и другой вариант — двойной щелчок на компоненте. Тогда компонент перенесется в центр формы, а затем вы можете его отбуксировать в нужное место.

Если вам надо разместить на форме несколько компонентов одного типа, то нажмите сначала клавишу `Shift` и, удерживая ее, щелкните на пиктограмме компонента на соответствующей странице библиотеки. После этого, отпустив `Shift`, щелкните несколько раз в разных местах формы. Каждый ваш щелчок будет размещать новый компонент. Чтобы прервать этот процесс, нажмите кнопку указателя со стрелкой, направленной по диагонали вверх и влево, в левой части палитры. Эту же кнопку вы можете нажать, если, выбрав компонент на странице, вы раздумали переносить его на форму.

Иногда вы знаете имя компонента, но не помните, на какой странице он расположен и не можете его найти. В этом случае вам может помочь команда `View | Component List`. При ее выполнении вам откроется диалоговое окно (рис. 2.20), содержащее алфавитный список всех компонентов. В нем вы можете найти нужный компонент по имени. В этом вам может помочь имеющееся сверху окно быстрого поиска по имени `Search by name`. По мере набора в нем первых символов имени указатель перемещается на соответствующие разделы списка. Выбрав нужный компонент, вы можете нажать кнопку `Add to form` внизу окна или сделать двойной щелчок на выбранном компоненте, и он перенесется на форму. Если вы не уверены, тот ли компонент нашли, нажмите клавишу `F1`, и вам будет показана справка по выделенному компоненту.

Рис. 2.20

Окно алфавитного списка всех компонентов



Удалить ошибочно перенесенный на форму компонент очень просто: выделите его и нажмите клавишу Delete.

Хороший стиль программирования

Приучите себя сразу при переносе компонента на форму изменять его имя Name, принятое по умолчанию. Имя должно быть осмысленным, чтобы потом, разбираясь в коде, вы легко могли бы понять, что означает та или иная функция, тот или иной компонент.

Настоятельно рекомендую следовать этому совету. Тем самым вы сэкономите себе много времени. Представьте себе простенькую форму, где имеется десяток кнопок, десяток меток и пара десятков разделов меню. Уверю вас, что если вы не даете компонентам осмысленные имена, то очень скоро вы начнете мучительно размышлять, что же это за кнопка **Button7**, щелчок которой вы обрабатываете в данной функции, и почему в нем задается какое-то значение надписи неизвестно где расположенной метки **Label3**. А уж на поиски таинственного раздела меню **N18** вы точно потратите немало времени. Так что любите себя, давайте компонентам понятные вам имена.

Перенеся компонент на форму, вы можете буксировать его в нужное место, можете изменять его размеры. Для этого выделите нужный элемент. Он станет окружен рамкой с маркерами. Потяните курсором за один из маркеров и размер компонента будет изменяться. При задержке курсора над размещенным на форме компонентом появляется ярлычок с его именем, а при нажатии кнопки мыши на компоненте, при его буксировке или изменении размеров появляется ярлычок с размерами компонента.

2.5.2.2 Родители и владельцы компонентов — Parent и Owner

Часто компоненты размещаются не непосредственно на формах, а на панелях, зрительно объединяющих группы компонентов по их назначению. В этом случае сначала на форме должны быть размещены панели, а потом на них располагаются компоненты.

Оконный компонент — форма, панель и т.д., включающий в себя как контейнер другие компоненты, выступает по отношению к ним как *родительский компонент*. У каждого компонента есть родитель. Им может быть форма или другой оконный компонент. В процессе выполнения приложения вы можете узнать родителя того или иного компонента по его свойству **Parent**. Это свойство можно читать и изменять только во время выполнения, в Инспекторе Объектов вы его не найдете.

Что дает понятие родительского компонента? Компонент может наследовать многие свойства своего родителя. Для всех визуальных компонентов вы можете

увидеть в Инспекторе Объектов такие свойства, как **ParentFont** и **ParentShowHint**, для оконных компонентов имеется еще свойство **ParentCtl3D**. Эти свойства указывают (если их значения установлены в **true**), что дочерний компонент наследует от родительского соответственно атрибуты шрифта, показа ярлычков, атрибуты своего оформления. Кроме того, значения свойств **Left** и **Top**, которые вы можете видеть в Инспекторе Объектов для любого визуального компонента и которые определяют положение левого верхнего угла компонента, измеряются в системе координат родительского компонента. Таким образом, например, при перемещении родительского компонента будут синхронно перемещаться и все его дочерние компоненты. Свойство **Anchors** определяет привязку дочерних компонентов к границам родительского компонента. В разделе 4.2.2 будет показано, как это свойство может обеспечивать изменение местоположения и размеров дочернего компонента при изменении границ родительского компонента.

Имеется еще два важных свойства, которые связывают дочерние компоненты с родительским. Это свойства **Visible** — видимый, и **Enabled** — доступный. Если в процессе выполнения приложения сделать в родительском компоненте **Visible** равным **false**, то станет невидимым не только родительский, но и все его дочерние компоненты. Аналогично, если в процессе выполнения приложения сделать в родительском компоненте **Enabled** равным **false**, то станут недоступными все его дочерние компоненты. Т.е. пользователь не сможет нажимать кнопки и производить любые другие действия в пределах данного родительского компонента.

Таким образом, понятие родительского компонента очень важное и его смысл выходит далеко за рамки просто эстетического оформления окна формы.

Отметим еще одно свойство компонентов, которое часто путается со свойством **Parent**. Это свойство **Owner** — *владелец данного компонента*. Свойство **Owner** устанавливается в момент создания компонента в процессе выполнения приложения. Владелец компонента — это тот компонент, при уничтожении которого (освобождении занимаемой им памяти) уничтожится и данный компонент. Этим и ограничивается связь между владельцем и компонентами, которыми он владеет, в отличие от множества указанных выше свойств, связывающих родительский и дочерние компоненты.

По умолчанию родителем и владельцем всех компонентов, размещенных на форме, является сама форма. Но если в процессе проектирования компонент размещается не непосредственно на форме, а на другом оконном компоненте, например, на панели, то родителем для него становится эта панель.

Все дочерние компоненты в оконном элементе располагаются в так называемой Z-последовательности. Для перекрывающихся компонентов (располагающихся друг на друге) Z-последовательность определяет, какой из них будет виден. Виден тот, который расположен в этой последовательности выше.

Обычно последовательность компонентов соответствует той, в которой они помещались на форму. Однако неоконные компоненты типа меток всегда лежат в Z-последовательности ниже любых оконных компонентов типа панелей и кнопок.

2.5.2.3 «Многослойное» размещение компонентов на форме

Рассмотрим теперь особенности размещения, связанные с обсуждавшимся выше понятием родительского компонента. Компоненты очень часто размещаются на панелях. Более того, нередко панели помещаются друг на друга, так что получается «многослойное» размещение компонентов на форме.

Если вы перенесли компонент из библиотеки не на форму, а на панель, то эта панель становится для него родительской. Вы никакими передвижениями не сможете переместить его за пределы родительской панели. Если же вы передумали размещать компонент на данной панели и хотите переместить его на другую панель или непосредственно на форму, то это можно сделать через буфер обмена **Clipboard**. Выделите курсором переносимый компонент и вырежьте его в **Clipboard**

командой Edit | Cut или «горячими» клавишами Ctrl-X. Затем щелкните на форме или на той панели, куда хотите перенести компонент, и выполните команду Edit | Paste, дублируемую «горячими» клавишами Ctrl-V. Компонент перенесется из Clipboard на новое место и обретет нового родителя — панель или форму.

Если вы в процессе проектирования перемещаете панель, она может накрыть какие-то компоненты, размещенные на форме или на другой панели. Будут ли при этом накрытые компоненты видны или не видны, определяется их местом в описанной ранее Z-последовательности. Неоконные компоненты типа меток заведомо будут невидимы, поскольку они располагаются в Z-последовательности всегда ниже любых оконных, к которым принадлежат панели. А вот видимостью других оконных компонентов — панелей, кнопок, окон редактирования можно управлять. Это делается командами меню Edit | Bring To Front и Edit | Send To Back. Первая из них перемещает выделенный оконный компонент на верх Z-последовательности и он начинает загораживать все другие оконные компоненты. А вторая команда перемещает выделенный оконный компонент на самый низ Z-последовательности и любые другие оконные компоненты, расположенные в Z-последовательности выше, начинают перекрывать его. Эти команды можно выполнить и проще: щелкнув правой кнопкой мыши и выбрав их из всплывающего меню.

Чтобы почувствовать все это, проведите эксперимент. Поместите на форму три панели: две из них непосредственно на форме, а третью — на одной из предыдущих. Разместите на форме и на панелях различные метки, кнопки и окна редактирования. Перемещайте компоненты по панелям, а сами панели — по форме, выполняйте команды Bring To Front и Send To Back и наблюдайте при этом видимость различных компонентов.

2.5.2.4 Поиск «пропавших» компонентов

Иногда бывает, что вы не можете найти на форме компонент, который, как вы знаете, на ней присутствует. Это бывает по нескольким причинам. Например, если вы используете метку типа **TLabel**, установив в ней свойство **AutoSize** (автоматическое изменение размера по размерам надписи) в **true** и стерев значение надписи **Caption**, то горизонтальный размер метки уменьшается до нуля и ее не будет видно на форме, пока во время выполнения приложения значение **Caption** не изменится. Компонент может «пропасть» также, если он накрыт другим компонентом, расположенным выше в Z-последовательности. Возможны и некоторые другие причины, например, такой выбор цветов, что компонент сливается с фоном.

Найти «пропавший» компонент можно, выбрав его имя в выпадающем списке, расположенном сверху окна Инспектора Объектов. Этот список содержит все компоненты, размещенные на форме. Если вы выберете в нем нужный компонент, то на форме вокруг него появится рамка с маркерами, видимая даже в случае, если компонент накрыт сверху какой-нибудь панелью или другим компонентом. При этом в Инспекторе Объектов станут видны страницы свойств и событий найденного компонента.

Если цель поиска заключалась в том, чтобы задать для компонента значения каких-то свойств или обработчики событий, то больше вам ничего и не надо. Если же вам все-таки надо добраться до компонента, накрытого другим компонентом или панелью, то придется или временно сдвигать куда-то эти помехи, или выполнить для них команду Send To Back, которая сдвинет их в низ Z-последовательности. По крайней мере, вы знаете, где расположен «беглец» и что надо сдвинуть, чтобы его найти.

Особо надо остановиться на вопросе, как, ничего не сдвигая, добраться до нижних панелей, если на них лежат другие панели, или как добраться до формы, если вся она накрыта панелями. Здесь возможен тот же подход, о котором говорилось выше. Но можно поступить проще. Выделите верхнюю панель и нажмите клавишу Esc. В окне Инспектора Объектов откроются страницы, связанные с ни-

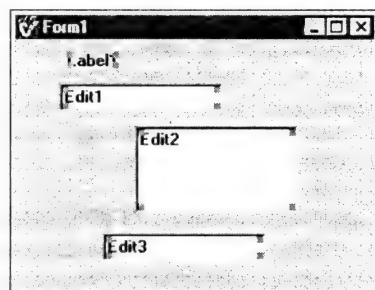
железащей панелью. Нажмите Esc еще раз и откроются страницы следующего слоя и т.д. до тех пор, пока не откроются страницы формы. Конечно, если у вас есть только один слой панели, то страницы формы откроются при первом же нажатии Esc.

2.5.2.5 Работа с группой компонентов, выравнивание компонентов по размеру и положению

На форме можно выделить группу компонентов, к которым применяется та или иная операция. Выделение группы возможно двумя способами. Если компоненты расположены непосредственно на форме и рядом друг с другом, то для выделения группы достаточно обвести курсором рамку вокруг них, и все они окажутся выделенными (рис. 2.21). Если компоненты группы расположены не непосредственно на форме, а, например, на панели, то выделить их рамкой невозможно. Так же невозможно выделить рамкой группу компонентов, расположенных в разных местах формы. В этих случаях выделение производится иначе. Выделяйте нужные компоненты курсором, нажав и не отпуская при этом клавишу Shift. Все компоненты, выделенные таким образом, войдут в группу.

Рис. 2.21

Выделенная группа компонентов



С выделенной группой компонентов можно производить следующие операции:

- Перемещать их одновременно, потянув курсором за один из выделенных компонентов. Это очень удобно, когда надо переместить группу компонентов на форме, не изменяя их взаимного расположения.
- Задавать в Инспекторе Объектов общие для всей группы свойства. При выделении группы на странице свойств в Инспекторе Объектов будут видны только их общие свойства. А индивидуальные свойства, такие, например, как имя компонента **Name** или надпись **Caption**, исчезнут. Вы можете задать особенности шрифта, оформления, цвет и т.п., которые будут присущи всем компонентам выделенной группы.
- Задать общий для всех компонентов группы обработчик какого-то события. Как вы увидите в процессе дальнейшего изучения приемов проектирования приложений, такая потребность возникает достаточно часто.
- Скопировать всю группу в буфер обмена Clipboard командой Edit | Copy или «горячими» клавишами Ctrl-C. После этого вы можете, например, открыть какую-то другую форму и перенести на нее группу из Clipboard командой Edit | Paste или «горячими» клавишами Ctrl-V. Это простой способ переносить фрагменты с одной формы на другую. Этот же прием удобно применять для переноса группы компонентов с одной панели на другую.
- Выравнивать компоненты группы по размеру и взаимному расположению. Как это делается мы сейчас рассмотрим.

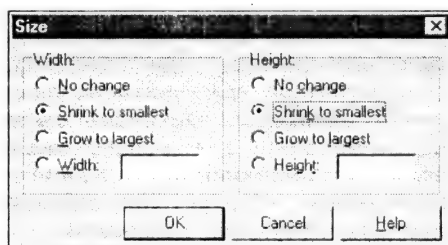
Когда вы размещаете компоненты на форме, трудно бывает добиться аккуратного вида окна, симметричного расположения компонентов, а иногда возникают проблемы и с тем, чтобы добиться одинакового размера их по горизонтали и верти-

кали. В C++Builder существует удобный инструмент для решения этих задач. Речь идет о командах **Edit | Align** — выравнивание размещения, **Edit | Size** — выравнивание размеров и **Edit | Scale** — масштабирование. Те же команды проще выбирать не из меню **Edit**, а из контекстного меню, которое всплывает, если вы щелкните правой кнопкой мыши на одном из компонентов группы (именно на компоненте, так как если вы щелкнете в стороне, то выделение группы снимется).

При выполнении команды **Size** — выравнивание размеров, вам открывается окно, представленное на рис. 2.22. Левая часть окна — **Width** устанавливает ширину компонентов. Вы можете выбрать варианты: **No change** — не изменять, **Shrink to smallest** — уменьшить до размера минимального из компонентов группы, **Grow to largest** — увеличить до размера максимального из компонентов группы, **Width** — задать в окне рядом с этой радиокнопкой ширину компонента в пикселях. Аналогичные варианты предлагаются в правой части окна для **Height** — высоты компонентов.

Рис. 2.22

Окно выравнивания размеров группы компонентов

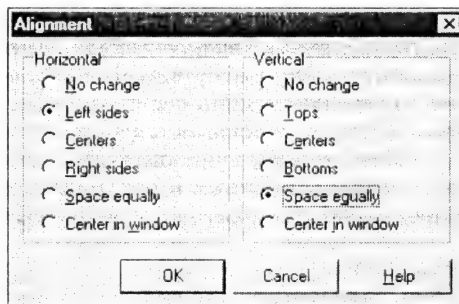


Например, на рис. 2.21 были изображены окна редактирования разных размеров. Если желательно все окна сделать одинаковыми, надо выделить их в группу, выполнить команду выравнивания размеров и для обоих измерений — ширины и высоты, указать, например, вариант **Shrink to smallest**.

При выполнении команды **Align** — выравнивание размещения, вам открывается окно, представленное на рис. 2.23. Левая часть окна — **Horizontal** устанавливает выравнивание компонентов по горизонтали. Вы можете выбрать варианты: **No change** — не изменять, **Left sides** — выровнять компоненты по их левым сторонам (т.е. левые стороны компонентов будут расположены друг под другом), **Centers** — выровнять компоненты по их центрам, **Right sides** — выровнять компоненты по их правым сторонам, **Space equally** — разместить с равными интервалами между компонентами, **Center in window** — расположить в центре окна.

Рис. 2.23

Окно выравнивания размещения группы компонентов



Правая часть окна — **Vertical** устанавливает выравнивание компонентов по вертикали. Тут имеются аналогичные варианты. Вы можете выбрать: **No change** — не изменять, **Tops** — выровнять компоненты по их верхним сторонам, **Center** — выровнять компоненты по их центрам, **Bottoms** — выровнять компоненты по их нижним сторонам, **Space equally** — разместить с равными интервалами по вертикали между компонентами, **Center in window** — расположить в центре окна.

Необходимо сделать некоторые уточнения по различным режимам выравнивания.

При выравнивании по границам компонентов (Left sides, Right sides, Tops, Bottoms, Center) на месте остается самый левый (выравнивание по горизонтали) или самый нижний (выравнивание по вертикали) компонент, а местоположение остальных подгоняется под него.

В режиме Space equally крайние компоненты (левый и правый при выравнивании по горизонтали и верхний и нижний при выравнивании по вертикали) не перемещаются. Получающиеся интервалы определяются положением этих крайних компонентов и числом промежуточных. Так что, пользуясь этим режимом, разместите сначала крайние компоненты так, как вам нужно, а уж затем используйте выравнивание. Выравниваются расстояния между верхними (при выравнивании по вертикали) или левыми (при выравнивании по горизонтали) границами компонентов. Поэтому, если выравниваемые компоненты имеют разные размеры, то расстояния между их примыкающими друг к другу краями будут не одинаковы. Компоненты могут даже накладываться друг на друга. Так что выравнивание по равным расстояниям имеет смысл только для компонентов равных размеров.

Режим Center in window не означает, что каждый компонент расположится в горизонтальном или вертикальном направлении в центре окна. В центре окна расположится только центр выделенной группы, а относительные сдвиги компонентов сохранятся неизменными. Кроме того учтите, что речь в данном случае идет не об окне формы, а о родительском окне компонентов группы. Если компоненты расположены, например, на панели, то подразумевается центр этой панели.

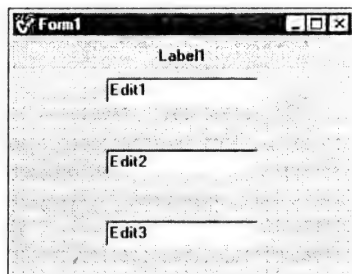
Посмотрим, как это все можно применить к примеру формы, показанной ранее на рис. 2.21. Пусть мы хотим, чтобы все компоненты — и окна редактирования и метка разместились по горизонтали в центре экрана, причем чтобы вертикальные расстояния между окнами редактирования были одинаковыми. И пусть мы уже выполнили описанное ранее выравнивание размеров окон редактирования.

Если выделить в группу все окна редактирования и задать по горизонтали выравнивание Center in window, а по вертикали — Space equally, то по вертикали окна действительно расположатся на равных расстояниях друг от друга, а ожидаемого выравнивания по горизонтали не будет. Вся группа окон сдвинется так, что ее центр будет совпадать с центром формы, но взаимные сдвиги окон останутся. Поэтому правильное выравнивание проводить в несколько приемов. Сначала по горизонтали задается Left sides, или Center, или Right sides — поскольку окна одинаковы, то все эти варианты эквивалентны. По вертикали при этом задается Space equally. В результате окна выстраиваются друг под другом с равными интервалами по вертикали. После этого надо повторно выделить эту группу окон и задать выравнивание по горизонтали Center in window, а по вертикали — No change. В заключение можно выделить одну метку и задать для нее аналогичное выравнивание: Center in window по горизонтали и No change по вертикали. В результате вы получите безупречное размещение, показанное на рис. 2.24.

Как видите, выравнивание компонентов часто требует многократного выполнения выравнивания в различных режимах. Даже с помощью команд всплывающего меню это делать неудобно. В C++Builder имеется гораздо более удобный инст-

Рис. 2.24

Окно, приведенное ранее на рис. 2.21, после выравнивания компонентов



румент выравнивания. Выполните команду View | Alignment Palette — палитра выравнивания. Появится окно палитры выравнивания, показанное на рис. 2.25. Назначение ее кнопок понятно из их пиктограмм. Верхний ряд кнопок относится к выравниванию по горизонтали и соответствует обсужденным выше вариантам выравнивания. Нижний ряд кнопок позволяет выравнивать по вертикали. Испытайте этот инструмент на каком-нибудь тестовом примере, и вы убедитесь в его эффективности. Чтобы полностью освоиться с выравниванием, добавьте на форму панели, разместите на них различные компоненты и попробуйте выровнять и сами панели и компоненты на них.

Рис. 2.25

Палитра выравнивания



Команда Edit | Scale позволяет пропорционально изменить масштаб всего расположенного на форме. Все размеры можно увеличивать или уменьшать вплоть до ста раз. В появляющемся диалоговом окне вам надо задать Scaling factor — масштабирующий коэффициент в %. Задав, например, 200, вы увеличите все компоненты в 2 раза.

2.5.2.6 Фиксация компонентов

После того, как вы тщательно разместили и выровняли компоненты, их местоположение полезно зафиксировать. Иначе в процессе последующей работы над проектом вы можете случайно сдвинуть тот или иной компонент, когда будете его выделять курсором, и всю работу по выравниванию придется начинать заново.

Чтобы этого не произошло, выполните команду Edit | Lock Controls. Она зафиксирует расположение всех компонентов на форме и не позволит их перемещать. Если в дальнейшем у вас все-таки возникнет потребность изменить расположение компонентов, то выполните повторно команду Edit | Lock Controls и компоненты будут разблокированы.

2.5.3 Инструментальные средства поддержки разработки кода

2.5.3.1 Применение Code Insight — Знатока Кода

Этот инструмент встроен в окно Редактора Кода и может оказать большую помощь при написании кода и его отладке. Он во многих случаях подскажет вам имена свойств, методов, событий, типы аргументов, типовые синтаксические конструкции и многое другое. Code Insight может работать в двух режимах: автоматическом и не автоматическом. В разделе 14.2.6 главы 14 вы можете посмотреть, как настроить Code Insight на автоматическую работу. Однако, автоматически возникающие подсказки очень полезны для начинающих, но могут раздражать более опытных пользователей. Поэтому имеется возможность отключить автоматический режим (он включен по умолчанию — см. раздел 14.2.6) и вызывать Code Insight по мере надобности, нажимая клавиши Ctrl-Shift-пробел или Ctrl-пробел в зависимости от того, к каким возможностям Code Insight вы хотите обратиться.

Code Insight! может выполнять следующие функции.

Завершение кода

Если вы написали в своем приложении имя компонента, поставили после него символы стрелки (->) и немного задержались с вводом последующего текста, то появится окно, содержащее список всех свойств, методов и событий класса, к которому принадлежит данный компонент. Вы можете выбрать из него требуемое

или начать писать первые символы свойства или метода, а затем нажать Enter, и в ваш код вставится соответствующее имя. Так будет при автоматической работе Code Insight. Если автоматический режим отключен, то вы можете вызвать ту же подсказку, если, поставив символы стрелки после имени компонента, нажмете Ctrl-пробел.

Если вы написали символ операции присваивания «=» и нажали Ctrl-пробел, то вам будет показан список возможных аргументов, совместимых по типу с переменной, которой будет присваиваться значение. Аналогичным образом можно получить подсказку по аргументам функций или процедур. Правда, возникающие списки подсказок в обоих этих случаях настолько длинные, что выбрать из него требуемое не так-то просто.

Параметры функций, процедур, методов

Если Code Insight работает в автоматическом режиме, то после того, как вы напишете имя функции или метода и поставите открывающуюся скобку, вы увидите список параметров и их типов. Причем, по мере того, как вы будете вводить значения аргументов, вам будет высвечиваться тип следующего параметра. Это, может быть, наиболее мощная возможность Code Insight, поскольку вряд ли кто-нибудь способен помнить параметры всех функций и методов C++Builder.

Если автоматическое высвечивание подсказок вы отключили, то можете вызвать подсказку, нажав клавиши Shift-Ctrl-пробел.

Шаблоны кода

В Code Insight занесено множество шаблонов стандартных структур языка C++. Причем вы сами можете добавлять или удалять эти шаблоны (см. раздел 14.2.6). Вызов шаблона производится нажатием клавиш Ctrl-J. Из выпадающего списка вы можете выбрать нужный шаблон. Например, если вы выбрали шаблон управляющей структуры **for**, то в ваш код занесется текст:

```
for ( ; ; )  
{  
  
}
```

Вам остается только заполнить этот шаблон, занеся в его заголовок соответствующие выражения, и написать тело цикла.

Оценка выражений

Эта способность Code Insight очень полезна в процессе отладки и подробнее будет рассмотрена позднее в разделе 2.6.3 при обсуждении способов отладки. Code Insight позволяет при останове или пошаговом выполнении приложения подвести курсор в окне Редактора Кода к имени любой переменной или к выражению и увидеть текущее значение оцениваемой величины.

Информация об идентификаторах — Code browser

Если задано автоматическое выполнение этого режима Code Insight, то при перемещении курсора мыши в тексте приложения над любой переменной автоматически высвечивается информация о ее объявлении, типе и о модуле и номере строки, содержащей это объявление. Это помогает при разработке больших приложений, но не очень удобно в простых задачах, так как на поиск этой информации Code Insight тратит заметное время. Так что можно рекомендовать обычно отключать эту возможность и включать ее только в случае необходимости.

Возможности Code browser существенно возрастают, если вы нажмете и не будете отпускать клавишу Ctrl. Тогда при перемещении курсора мыши над любым идентификаторами кода идентификатор выделяется цветом и подчеркиванием, а курсор приобретает вид руки. Если вы щелкнете на выделенном идентификаторе,

перед вами в окне Редактора Кода откроется файл, содержащий объявление соответствующего класса, свойства, метода, переменной и курсор перейдет к строке этого объявления. Такая возможность увидеть объявление любого идентификатора очень помогает во многих случаях разработки кода. Причем эта возможность не зависит от того, включено или выключено автоматическое выполнение Code browser.

Поиск информации об идентификаторах Code browser проводит в каталогах, устанавливаемых при настройке C++Builder при выполнении команд Project | Options (на странице Directories/Conditionals) и Tools | Environment Options (на странице Library).

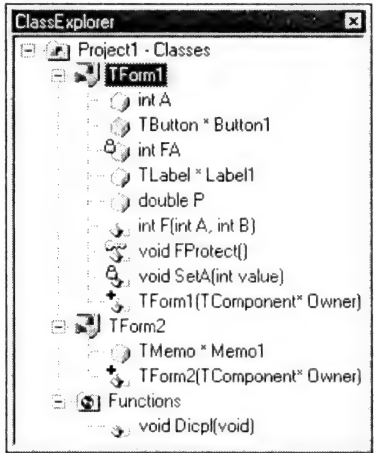
Code browser не может находить информацию об идентификаторах, объявленных в новых, еще не сохраненных модулях.

2.5.3.2 Исследователь Классов ClassExplorer

Исследователь Классов ClassExplorer показывает дерево всех типов, классов, свойств, методов, глобальных переменных и глобальных функций, содержащихся в модуле, открытом в Редакторе Кода.

По умолчанию окно Исследователя Классов (см. рис. 2.26) появляется автоматически встроенным в окно Редактора Кода (см. рис. 2.4 б). Правда, это поведение по умолчанию может быть изменено отключением опции Automatically show Explorer на странице ClassExplorer при выполнении команды Tools | Environment Options. В этом случае при необходимости вы можете вызвать Исследователя Кода командой View | ClassExplorer. Как указывалось в разделе 2.2.8, окно очень удобно встраивать на отдельную страницу в окне Инспектора Объектов.

Рис. 2.26
Окно Исследователя Классов ClassExplorer



В окне Исследователя Классов вы можете видеть структуру своего проекта, модулей и классов. На рис. 2.26 приведен пример, в котором встречается большинство используемых при этом значков: проекта (**Project1**), формы (**TForm1**, **TForm2**), опубликованного (**published**) свойства (**A**), опубликованных и открытых (**public**) данных (объектов — **Button1**, **Label1**, переменных — **P**), открытых методов (**F**), закрытых (**private**) данных (**FA**) и методов (**SetA**), защищенных (**protected**) методов (**FProtect**), конструкторов (**TForm1**). В Исследователе Классов даны также сведения и функциях, объявленных вне класса (**Displ**). Правда, учтите, что сведения даются только о тех функциях, для которых в заголовочном модуле или в модуле реализации имеется прототип. Функции, просто описанные в файле реализации, в Исследователе Классов не отображаются.

Если вы щелкнете в окне Исследователя Классов на имени переменной или функции, то курсор в окне Редактора Кода перейдет на строку, в которой эта пере-

объявление. Если же объект определен в каком-то другом, в частности, в системном модуле, то в окно Редактора Кода будет загружен соответствующий модуль и уже в нем курсор расположится на строке, содержащей объявление. После того, как вы получили требуемую информацию, вы можете выгрузить посторонний модуль из окна Редактора Кода командой контекстного меню *Close Page*.

Если вы поместите курсор на имени заголовочного файла, подключаемого к проекту директивой **#include**, щелкнете правой кнопкой мыши и во всплывающем меню выберете раздел *Open File at Cursor*, то в окно Редактора Кода будет загружен соответствующий файл и вы сможете посмотреть объявления различных содержащихся в нем функций, констант, макросов и т.п.

Теперь рассмотрим возможности навигации в коде и использование закладок. Вы можете пометить закладками какие-то операторы в разных частях кода и затем быстро перемещаться между этими частями. Чтобы сделать закладку, надо установить курсор в нужной строке, щелкнуть правой кнопкой мыши и выполнить команду *Toggle Bookmarks*. Откроется список возможных закладок, как показано на рис. 5.15. В нем вы можете пометить закладку, которую хотите привязать к данной строке, щелкнув на ней левой кнопкой мыши. Если вы хотите удалить какую-то ранее введенную закладку, щелкните на ней правой кнопкой мыши.

Когда вам потребуется в дальнейшем в процессе работы над приложением вернуться к введенной закладке, вы аналогичным образом можете выполнить команду *Goto Bookmarks* и в аналогичном списке закладок выбрать нужную.

Если вам нужны закладки, чтобы параллельно наблюдать два различных фрагмента кода, например, чтобы перенести или скопировать какие-то операторы из одного фрагмента в другой, то удобно открыть второе окно редактирования. Это можно сделать командой контекстного меню *New Edit Window*. Тогда вы можете в одном окне перейти к одной из закладок, в другом — к другой и затем одновременно работать с обоими фрагментами кода.

2.5.3.4 Справочная система C++Builder и программа ее конфигурирования OpenHelp

Справка в C++Builder может вызываться из меню *Help*. Это меню имеет, в частности, разделы:

C++Builder Help	вызов справки по C++Builder и C++
C++Builder Tools	вызов справок по инструментарию C++Builder 5
Windows API/SDK Help	вызов справок по Windows

Помимо этого в меню включен ряд разделов получения информации через Интернет.

Справку можно получить не только из меню *Help*, но и с помощью контекстно-зависимого поиска практически из любого окна C++Builder. Вы можете выделить на форме какой-то компонент, нажать *F1* и вам будет показана тема справки, связанная с этим компонентом. Если вы, находясь в окне Редактора Кода, установите курсор на имени какой-то функции, свойства или метода какого-то компонента и нажмете *F1*, то вам также будет показана справка по интересующему вас вопросу. Аналогично можно получить контекстную справку о свойстве компонента из окна Инспектора Объектов, выделив соответствующее свойство.

Правда, к сожалению, изредка такой контекстный поиск не дает правильный результат. Иногда вы в ответ получаете сообщение, что такой темы нет, и совет обратиться к разработчикам программы. А иногда просто вы попадаете совсем не на ту тему.

В подобных случаях можно посоветовать выходить на требуемую тему через страницу справки *Содержание*. Если вам требуется информация о компоненте, свойстве, методе, событии, то наиболее удобно раскрыть на этой странице книжку

«Visual Component Library Reference» (Обзор библиотеки визуальных компонентов), затем раскрыть «Alphabetical Object and Component Listing» (Алфавитный список объектов и компонентов) и в этом списке отыскать по алфавиту требуемый компонент. А из окна справки компонента всегда можно найти все его свойства, методы и события.

Если вам нужно найти справку по функциям, объявленным в библиотеке компонентов, и обычный контекстный поиск не помогает, то удобно на странице справки Содержание открыть книжку «Visual Component Library Reference» (Обзор библиотеки визуальных компонентов), затем раскрыть «Alphabetical Routines Listing» (Алфавитный список функций) или «Categorical Routines Listing» (Список функций по категориям), а затем найти нужную функцию в соответствующем разделе.

Если вам нужно найти справку по функциям C, то удобно на странице справки Содержание открыть книжку «C Runtime Library Reference» (Обзор библиотеки C), затем раскрыть «Categorical Routines and Types Listing» (Список функций и типов по категориям) или «Alphabetical Routines and Types Listing» (Список функций и типов по алфавиту), а затем найти нужную функцию в соответствующем разделе.

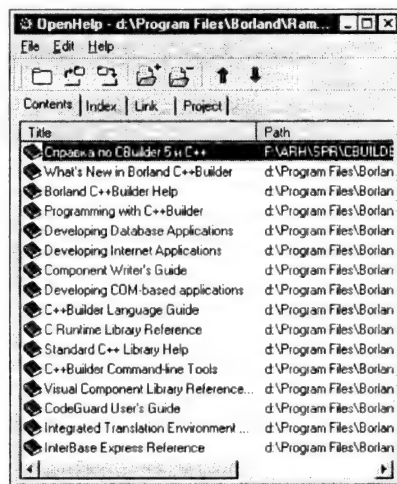
Для получения справки по API Windows, по сообщениям Windows и т.п. полезно открыть в окне справки один из файлов ...\program files\Common Files\Borland\Borland Shared\MShelp\95guide.hlp или ...\program files\Common Files\Borland Shared\MShelp\api.hlp. Можно также воспользоваться разделом Windows API/SDK Help меню Help.

А теперь рассмотрим очень интересный инструмент, позволяющий проводить настройку справочной системы. Речь идет о программе Borland OpenHelp, вызываемой командой Help | Customize. Окно этой программы показано на рис. 2.28. OpenHelp предоставляет вам простой путь конфигурирования файлов справки .hlp. При этом можно добавлять и убирать файлы справки, которые будут появляться в таблице содержания и в предметном указателе справки. В частности, можно встроить в систему собственные справочные файлы на русском языке. Например мы с группой соавторов создали русскую справку по C++Builder 5 и C++ (на рис. 2.28 изображен момент встраивания ее в C++Builder), которую я сам с удовольствием использую в своей работе. Ее эскизная версия содержится на диске, прилагаемом к данной книге. Полноценная версия справки, как я надеюсь, будет выпущена в серии книг «Все о C++Builder» и вы сами сможете убедиться, что встраивание собственных справок в C++Builder очень удобно.

OpenHelp хранит информацию о справочной системе в проекте. Файл этого проекта имеет расширение .ohp и хранится в каталоге /Help. Вы можете изменить

Рис. 2.28

Окно OpenHelp с открытой страницей Contents



состав справочной системы: таблицы Содержание (Contents), таблицы Предметный Указатель и контекстной справки, доступ к которой осуществляется из ИСП C++Builder клавишей F1.

OpenHelp позволяет также удалить ссылки системного реестра на устаревшие файлы справки. Дело в том, что нередко системный реестр и файл WINHELP.INI загромождаются ссылками на устаревшие файлы справок. Вы можете быстро очистить от них реестр, выполнив в окне OpenHelp команду File | Clean Registry.

Ниже изложена методика модификации справочной системы.

Таблица Содержание хранится в файле с расширением **.toc**, подобном файлам содержания Windows **.cnt**, только без предложений **Include**. Чтобы добавить файлы в таблицу Содержание, надо сделать следующее:

1. Перейти в окне OpenHelp на страницу Contents (см. рис. 2.28).
2. Выполнить команду Edit | Add Files.
3. Выбрать или написать имена одного или более добавляемых файлов **.toc** или **.cnt**.
4. Щелкнуть на ОК.
5. Вы можете переместить файл на желательное вам место среди других файлов. Для этого выделите файл и переместите его, пользуясь кнопками со стрелками на странице Contents.
6. Выполнить команду File | Save Project или File | Save Project As.

Для удаления файла из таблицы Содержание надо:

1. Перейти в окне OpenHelp на страницу Contents.
2. Выделить удаляемые файлы.
3. Выполнить команду Edit | Remove.
4. Выполнить команду File | Save Project или File | Save Project As.

Таблица Предметный Указатель хранит ссылки на файлы справок **.hlp**. Чтобы добавить файлы в таблицу Предметный Указатель или удалить файлы из нее надо произвести те же операции, которые были рассмотрены выше, но только работать на странице Index и добавлять или удалять файлы **.hlp**.

Контекстно-зависимый поиск справки состоит из файлов **.hlp**, доступных с помощью так называемых макросов **ALink**. Эти макросы используются в C++Builder при нажатии клавиши F1 в Инспекторе Объектов, Редакторе Кодов, палитре компонентов. Чтобы добавить файлы в контекстно-зависимый поиск, надо произвести те же операции, которые описаны выше, но на странице Link.

2.6 Отладка приложений

2.6.1 Компиляция и компоновка проекта

Мастерство программиста — разработчика приложения определяется вовсе не его умением писать безошибочные программы (написать сложную программу без ошибок не может никто). Мастерство определяется умением быстро, эффективно и надежно отлаживать и тестировать свое приложение. Вопросы тестирования очень важны, но они относятся к проблемам программирования и не связаны с тематикой данной книги. А вот техническими возможностями отладки приложений в ИСП C++Builder мы сейчас займемся, поскольку ими должен хорошо владеть каждый разработчик.

Тем, кто пока слабо владеет программированием и не знает языка C++, изложенный в разделе 2.6 материал сначала стоит просто просмотреть «по диагонали», чтобы уловить главное. А после более тесного знакомства с C++ я бы рекомендовал вернуться к этому материалу и познакомиться с ним более осмысленно.

Компиляция приложения может выполняться несколькими способами. Если вы работаете с группой проектов (см. раздел 2.4.3), то все, описанное далее, относится к тому проекту, который в данный момент активен.

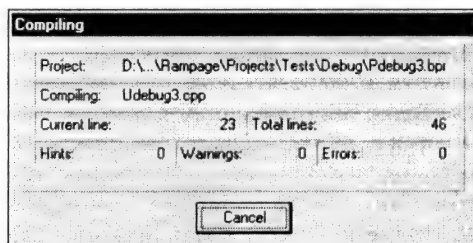
Компиляция с последующим выполнением приложения осуществляется командой Run | Run, или соответствующей быстрой кнопкой, или «горячей» клавишей F9. В этом случае производится компиляция программы, ее компоновка, создается выполняемый модуль .exe и он запускается на выполнение. Впрочем, создание модуля .exe и выполнение будет проводиться только в случае, если при компиляции и компоновки не обнаружены неисправимые ошибки.

В процессе компиляции и компоновки на экране появляется окно, приведенное на рис. 2.29. В его верхней строке вы видите имя компилируемого проекта. В следующей строке отображается текущая операция: компиляция определенного модуля (на рис. 2.29 показан момент компиляции модуля **Udebug3.cpp**) или компоновка (Linking). В третьей строке окна отображается текущая строка модуля (Current line), обрабатываемая компилятором, и общее число строк в модуле (Total lines). В нижней строке отображается обнаруженное на данный момент число замечаний (Hints), предупреждений (Warnings) и ошибок (Errors). Клавиша Cancel внизу окна позволяет прервать процесс компиляции и компоновки.

Если в компилируемом файле встретились неисправимые ошибки, выполняемый файл не будет создан. Если ошибок нет, файл создастся, но и в этом случае у компилятора могут быть предупреждения и замечания, которые вам надо внимательно изучить. Как это сделать — будет сказано позднее.

Рис. 2.29

Окно компиляции и компоновки



При компиляции проекта, состоящего из нескольких модулей, компилируются только те модули, тексты которых были изменены с момента предыдущей компоновки проекта. Это существенно экономит время компиляции.

При выполнении команды Run вы можете задать командную строку, если ваше приложение предусматривает передачу в него каких-то параметров. Для этого надо сначала выполнить команду Run | Parameters и в открывшемся окне написать требуемую командную строку.

Не всегда вам надо компилировать проект и тут же выполнять его. Часто вам важнее просто проверить, не содержат ли ваши последние изменения кода каких-то ошибок. В этом случае вам не имеет смысла терять время на выполнение проекта и лучше воспользоваться другими командами меню: Project | Compile Unit, Project | Make Project или Project | Build Project.

Команда Compile выполняет компиляцию только того модуля, который выделен вами в окне Редактора Кода или в Менеджере Проектов. Эта команда позволяет наиболее быстро проверить наличие ошибок или замечаний при компиляции модуля, так как не осуществляется компоновка программы и не компилируются никакие другие модули. Если компиляция прошла успешно, создается объектный файл .obj откомпилированного модуля.

Команда Make выполняет компиляцию всех тех модулей, тексты которых были изменены с момента предыдущей компоновки проекта. Если компиляция прошла успешно, то создаются объектные файлы модулей .obj и осуществляется компоновка программы. Если и она прошла успешно, то создается выполняемый

модуль .exe. Таким образом, отличие Make от Run только в том, что после компоновки не производится выполнение приложения.

Команда Build подобна команде Make за одним исключением — компилируются все модули, независимо от того, когда они в последний раз изменялись. Конечно, выполнение этой команды требует наибольшего времени. Но иногда только ее и можно использовать. Например, если с момента последней компиляции вы ничего не изменяли в модулях, но хотите откомпилировать проект с новыми опциями компилятора, например, изменить уровень оптимизации. Тогда все иные команды, кроме Build, не произведут повторной компиляции. Так что эта команда во многих случаях необходима.

Помимо описанных команд компиляции имеется еще две: Project | Make All Projects и Project | Build All Projects. Они подобны рассмотренным командам Make и Build, но при работе с группой проектов (см. раздел 2.4.3) относятся не к одному, а ко всем проектам группы.

По умолчанию все команды компиляции в C++Builder 5 выполняются в фоновом режиме. Эта новая возможность, введенная в C++Builder 5, позволяет осуществлять во время компиляции и компоновки любые другие работы в ИСР. Это, конечно, удобно, но не всегда. Дело в том, что фоновая компиляция осуществляется медленнее. Кроме того, по завершении компиляции в фоновом режиме окно компилятора исчезает и при этом не показываются результаты компиляции: прошла ли она успешно, или имеются замечания. Поэтому, если у вас нет каких-то работ в ИСР, которые можно выполнять во время компиляции, лучше отключить фоновый режим компиляции. Это можно сделать, выполнив команду Tools | Environment Options и выключив на странице Preferences опцию Background Compilation (подробнее об опциях см. в разделе 14.2.10).

Если фоновый режим компиляции отключен, то после окончания компиляции рассмотренными командами (кроме Run) в окне рис. 5.17 во второй строке появляется одно из трех итоговых сообщений: «Done: Make» — «Результат: выполнено», «Done: There are errors» — «Результат: имеются ошибки», «Done: There are warnings» — «Результат: имеются предупреждения».

2.6.2 Сообщения компилятора и компоновщика

В предыдущем разделе рассмотрены команды компиляции проекта. Теперь давайте посмотрим, какие сообщения об ошибках и какие предупреждения выдает компилятор и как на них надо реагировать. Давайте для этого сделаем простое приложение с ошибочными операторами. Начните новое приложение, перенесите на форму метку **Label** и кнопку **Button**. В обработчик щелчка кнопки введите следующие операторы:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int i,j;
    double A;
    for(i = 0; i < 50; i++)
        A *= 10000;           // увеличение A в 10000 раз
    Label1->Caption = "A = " + B;
}
```

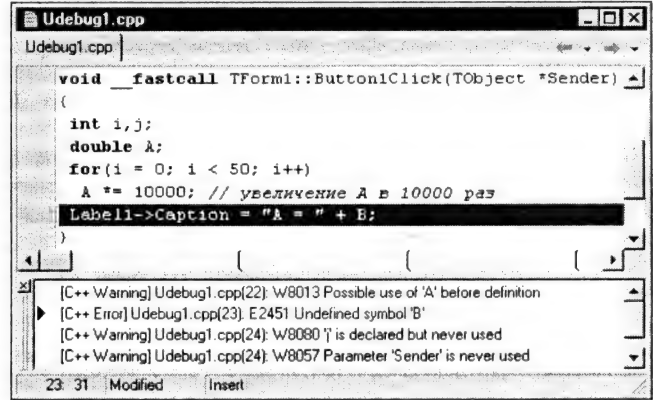
Этот код содержит цикл **for**, который выполняется 50 раз, увеличивая в каждой итерации переменную **A** в 10000 раз.

Попробуйте выполнить команду Run | Run, или нажать соответствующую быструю кнопку, или нажать клавишу F9. Вы увидите в окне Редактора Кода картину, показанную на рис. 2.30. Впрочем, такую картину вы увидите, если в окне опций проекта, вызываемом командой Project | Options на странице Compiler в группе опций Warnings включена опция All. Это означает, что компилятор должен отображать все

свои замечания. Этот режим наиболее удобен для отладки. Поэтому если у вас сообщений меньше, чем показано на рис. 2.30, проверьте, включена ли у вас указанная опция. Подробнее об опциях компиляции вы можете узнать, ознакомившись с разделом 14.2.9.

Рис. 2.30

Окно Редактора Кода с сообщениями о замечаниях и ошибках



Внизу окна видны сообщения о замечаниях и ошибках. Прежде, чем рассматривать их, отметим, что они размещены во встраиваемом окне сообщений. Как всякое встраиваемое окно, вы можете вынуть его из окна Редактора Кода и сделать самостоятельным или встроить, например, в окно Инспектора Объектов. Впрочем, это окно достаточно удобно сохранить именно встроенным в Редактор Кода. Места оно занимает немного. Вы можете также увеличить или уменьшить его размер до желаемого. Если вы вынули окно сообщений из окна Редактора Кода, а при дальнейших манипуляциях с окнами «потеряли» (не видите, поскольку оно загорожено другими окнами), или если вы закрыли это окно его системной кнопкой с крестиком, то вы можете щелкнуть в окне Редактора Кода правой кнопкой мыши и выбрать в контекстном меню команду Message View.

Теперь рассмотрим полученные сообщения компилятора. Первое сообщение:

```
[C++ Warning] Udebug1.cpp(22): W8013 Possible use of 'A' before definition
([C++ Предупреждение] модуль Udebug1.cpp, строка 22: W8013 Переменная 'A', возможно, используется до того, как ей присвоено значение)
```

Это предупреждение о том, что вы не инициализировали переменную **A** и ее значение к моменту первого выполнения оператора в строке 22 не определено. Чтобы узнать, что это за строка, достаточно сделать мышью двойной щелчок на этом предупреждении. Тогда в окне Редактора Кода выделится соответствующая строка текста. В данном случае вы увидите, что речь идет об операторе

```
A *= 10000;           // увеличение A в 10000 раз
```

Действительно, мы ввели локальную переменную **A** и нигде не задали ее начальное значение. С точки зрения компилятора это не ошибка, а Warning — замечание. Но для программы это действительно логическая ошибка, так как неизвестно, чему будет равно значение этой переменной при входе в цикл. Если мы хотим, например, чтобы при каждом выполнении функции значение **A** равнялось 1, мы должны изменить объявление переменной:

```
double A = 1;
```

или добавить перед циклом оператор:

```
A = 1;
```

Если же мы хотим, чтобы в переменной **A** накапливался результат при каждом щелчке на кнопке, мы должны убрать ее объявление из функции и сделать переменную **A** глобальной, инициализировав ее в объявлении приведенным ранее оператором.

Этот пример показывает, что не надо пренебрегать замечаниями компилятора. Всегда лучше перед запуском на выполнение сначала только откомпилировать проект и тщательно проанализировать каждое замечание. Только исправив все, что требует исправления, имеет смысл выполнять приложение. Это сэкономит вам время, которое в противном случае вы потратите на поиск причин, по которым ваша программа работает неправильно.

Хороший стиль программирования

Просматривайте все замечания компилятора и стремитесь найти и устранить причины, вызвавшие эти замечания. Игнорируя их, вы рискуете снизить надежность и эффективность своего приложения.

К сожалению, предупреждение об отсутствии инициализации компилятор дает только для локальных переменных. Если вы забыли инициализировать глобальную переменную, компилятор этого не заметит и вы получите в программе логическую ошибку, которую придется отлаживать в процессе отладки. Бойтесь таких ошибок. Поскольку значение неинициализированной переменной неопределенно, то ваша программа при разных запусках и на разных компьютерах может выдавать разные результаты.

Вернемся теперь к рис. 2.30. Второе сообщение компилятора:

```
[C++ Error] Udebug1.cpp(23): E2451 Undefined symbol 'B'.
([C++ Ошибка] модуль Udebug1.cpp, строка 23: E2451 Необъявленный идентификатор 'B')
```

Это уже сообщение об ошибке. В данном случае в операторе

```
Label1->Caption = "A = " + B;
```

вместо переменной **A** мы указали переменную **B**, которая не была объявлена. Строка кода с этой ошибкой выделена в окне Редактора Кода и курсор остановился около необъявленного идентификатора. Ошибки такого рода будут у вас чаще всего — это результат описки в имени переменной, компонента, свойства, метода, функции.

Поскольку рассмотренная выше ошибка неисправима, то в данном примере выполняемый модуль не формируется и приложение не выполняется.

Третье сообщение компилятора:

```
[C++ Warning] Udebug1.cpp(24): W8080 'j' is declared but never used.
([C++ Предупреждение] модуль Udebug1.cpp, строка 24: W8080 Переменная 'j' объявлена, но нигде не используется)
```

Действительно, мы объявили переменную **j**, но не использовали ее. Если эта переменная — заготовка для каких-то будущих процедур, то это замечание можно проигнорировать. Но если переменная **j** действительно не нужна, то ее объявление лучше удалить из текста, так как под эту переменную тратится, конечно, небольшой, но совершенно бессмысленный объем памяти.

Последнее сообщение компилятора:

```
[C++ Warning] Udebug1.cpp(24): W8057 Parameter 'Sender' is never used
([C++ Предупреждение] модуль Udebug1.cpp, строка 24: W8057 Параметр 'Sender' нигде не используется)
```

В обработчик событий действительно передается параметр **Sender**, как вы можете видеть в заголовке процедуры. Этот параметр — компонент, в котором произошло событие. Нам он не нужен в данном обработчике, так что это предупреждение можно проигнорировать.

Итак, наш пример не откомпилировался из-за ошибки с использованием не-объявленной переменной **В**.

Если вы исправите в ошибочном операторе переменную **В** на **А**, то, увы, получите новое сообщение об ошибке:

```
[C++ Error] Udebug1.cpp(23): E2060 Illegal use of floating point.  
([C++ Ошибка] модуль Udebug1.cpp, строка 23: E2060 Недопустимое  
использование плавающей точки)
```

Действительно, в правой части оператора

```
Label1->Caption = "A = " + A;
```

складывается строка «A = » и переменная с плавающей точкой **А**. Подобные ошибки использования несовместимых типов тоже очень распространены и вы, вероятно, не раз невольно будете их делать.

Правильный оператор в нашем примере должен иметь вид:

```
Label1->Caption = "A = " + FloatToStr(A);
```

Итак, после всех исправлений ваш код может иметь вид:

```
double A = 1;  
void __fastcall TForm1::Button1Click(TObject *Sender)  
{  
    int i;  
    for(i = 0; i < 50; i++)  
        A *= 10000;           // увеличение A в 10000 раз  
    Label1->Caption = "A = " + FloatToStr(A);  
}
```

Теперь откомпилируйте ваше приложение и выполните его.

2.6.3 Что делать, если произошла ошибка выполнения

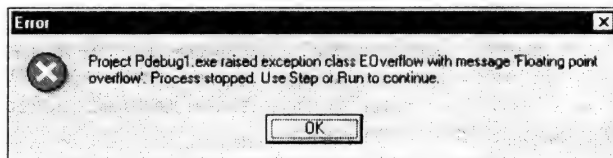
Если ваше приложение откомпилировалось и стало выполняться, это, увы, еще не означает, что оно правильно работает. В нем может быть еще множество ошибок времени выполнения. Это могут быть постоянные логические ошибки, проявляющиеся всегда при выполнении некоторых частей вашей программы. Это могут быть ошибки, проявляющиеся только при каких-то сочетаниях данных: ошибки деления на нуль, переполнения, открытия несуществующего файла и т.п. Наконец, могут быть случайные перемежающиеся ошибки, когда одна и та же задача иногда выполняется нормально, а иногда — нет. Такие ошибки, которые наиболее трудно вылавливать, связаны обычно с отсутствием инициализации переменных в каких-то режимах работы. В этом случае выполнение приложения зависит от случайного состояния памяти компьютера.

Во всех подобных случаях причины ошибок выявляются в процессе отладки. Наше приложение тоже не свободно от ошибок времени выполнения. Запустите его на выполнение и щелкните на кнопке. Все вроде бы сработает нормально. Но попробуйте повторно щелкнуть на той же кнопке. Выполнение прервется и перед вами возникнет окно, приведенное на рис. 2.31.

Перевод текста в этом окне гласит: «Проект Pdebug1.exe вызвал генерацию исключения класса EOverflow с сообщением 'Переполнение при операции с плавающей запятой'. Процесс остановлен. Используйте команды Step или Run для продолжения».

Рис. 2.31

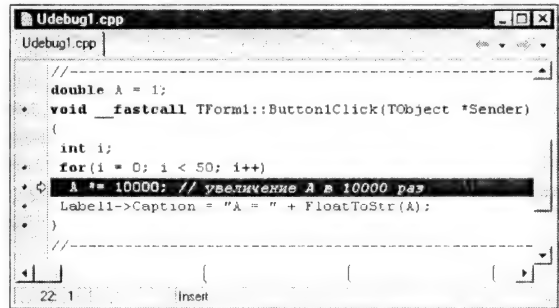
Сообщение отладчика об ошибке переполнения



Это сообщение об ошибке, приведшей к генерации так называемого исключения. Исключения (exceptions) генерируются при различных ошибках — исключительных ситуациях. Способы работы с исключениями вы можете посмотреть в главе 12 в разделе 12.10. А сейчас перед нами стоит вопрос: Что делать? Щелкнув на кнопке ОК, вы попадете в окно Редактора Кода и увидите в нем (рис. 2.32) код вашей программы с выделенной строкой, около которой стоит зеленая стрелка. Это тот оператор, при выполнении которого произошла ошибка.

Рис. 2.32

Окно Редактора Кода с выделенной строкой, в которой произошла ошибка



Дальнейшие ваши действия сводятся к одной из следующих альтернатив.

- Можно нажать клавиши Ctrl-F2 и тем самым прервать выполнение и отладку приложения.

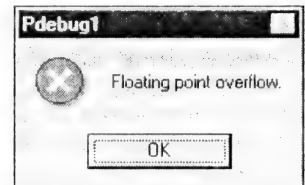
Это, конечно, возможный вариант действий, если вы уже поняли, в чем заключается ошибка, и готовы ее исправить. Если же нет, то вам придется анализировать ситуацию, обдумывать возможные варианты ее исправления и т.д. Берегите свое время и серые клеточки своего мозга! Если неясно, в чем заключалась ошибка, прервавшая выполнение приложения, то не тратьте силы на размышления. Прежде, чем прерывать сеанс работы с приложением, надо получить дополнительную информацию о состоянии переменных, т.е. провести отладку. А потом уже можно обдумывать ситуацию, имея на руках данные.

- Можно выполнить команду Run | Run (или нажать соответствующую быструю кнопку, или клавишу F9), чтобы попытаться, несмотря на ошибку, продолжить вычисления.

При этом перед вами возникнет окно (рис. 2.33) с сообщением о виде ошибки, после чего вы можете продолжать работать с приложением. Но это ни к чему не приведет, так как при очередном щелчке на кнопке ситуация с ошибкой повторится.

Рис. 2.33

Окно информации об исключении



- Вы можете пройти часть программы по шагам, как рассмотрено в одном из следующих разделов.

Но прежде, чем это делать, вам надо получить какую-то информацию. Иначе такой проход ничего вам не даст.

Итак, единственно правильный способ действий, если причина ошибки неясна:

- Надо получить информацию о происходящих в приложениях процессах, приведших к ошибке.

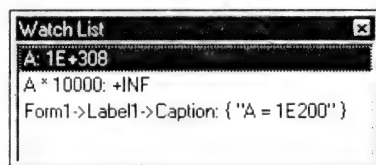
Это можно сделать несколькими способами. В ИСР C++Builder 5 имеется Мастер оценки выражений — ToolTip Expression Evaluation. Подведите курсор мыши к имени одной из переменных в коде, например, к **A**, и увидите текст: «**A=1E+308**». Таким простым способом вы можете узнать значения переменных программы в данный момент. Правда, иногда могут возникать трудности. Некоторые переменные не удается наблюдать, потому что оптимизирующий компилятор удалил их из результирующего кода.

2.6.4 Окно наблюдения Watch List

Мастер оценки выражений, конечно, хороший инструмент, но он дает значения только отдельных переменных, а в сложных приложениях вам надо бы иметь перед глазами значения нескольких переменных сразу, чтобы из их сравнения понять причины неправильной работы приложения. Такую возможность предоставляет вам окно наблюдения Watch List. Сделать его видимым можно командой View | Debug Windows | Watches. Но этого даже можно не делать. Достаточно подвести курсор в коде к интересующей вас переменной и нажать Ctrl-F5. При этом окно \pord plain наблюдения автоматически откроется и в нем появится имя переменной и ее значение (значение переменной будет видно только при остановке выполнения приложения и переходе в ИСР C++Builder). Затем вы можете подвести курсор к другой переменной, опять нажать Ctrl-F5 и в окне наблюдения появится новая строка. Более того, вы можете выделить курсором какое-то выражение, нажать Ctrl-F5 и в окне наблюдения увидеть значение этого выражения (рис. 2.34).

Рис. 2.34

Окно наблюдения



На рис. 2.34 представлено окно наблюдения в момент, когда в нашем приложении произошла генерация исключения. Можно видеть, что переменная **A** равна 10^{380} , а значение выражения **A * 10000** указано равным **+INF**. Этот идентификатор соответствует плюс бесконечности. Значит, при очередном умножении **A** на 10000 возникает переполнение, так как переменная типа **double** не может хранить столь большое число. В этом и заключается причина возникновения ошибки в нашем примере.

Окно наблюдения прекрасный инструмент, но при его использовании надо соблюдать некоторые правила. Когда вы пишете код, то по умолчанию считается, что все компоненты принадлежат текущей форме и все свойства, для которых вы не указываете объект, относятся также к текущей форме. В окне наблюдений подобных предположений не делается. Поэтому, если вы занесете в окно, например, выражение **Label1->Caption**, то при остановке выполнения приложения получите вместо значения этого выражения сообщение: «Undefined symbol 'Label1'» — не определен символ 'Label1'. Если же вы добавите в это выражение ссылку на форму **Form1**, как это сделано на рис 2.34, то все будет работать нормально.

Иногда переменные не удается наблюдать, потому что оптимизирующий компилятор удалил их из результирующего кода и помещает соответствующие значения в системные регистры. Это ускоряет выполнение вычислений в приложении, экономит память, но препятствует наблюдению переменных в процессе отладки. В таких случаях можно объявить соответствующую переменную с ключевым словом **volatile**. Например:

```
volatile int x;
```

Спецификатор **volatile** говорит компилятору, что данную переменную нельзя хранить в регистре.

Другой (и лучший) вариант достичь той же цели — выполнить команду Project | Options и в открывшемся окне на странице Advanced Compiler выключить опцию Register Variables (см. подробнее в разделе 14.2.9.4).

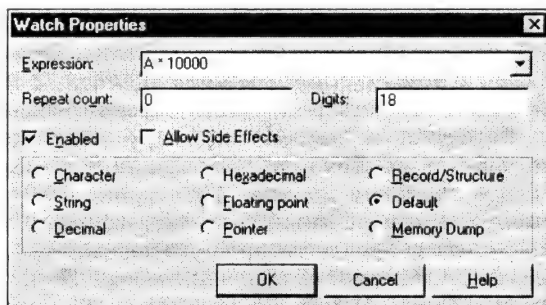
Только имейте в виду, что после окончания отладки оба эти варианта лучше убрать, чтобы не снижать эффективность своего приложения.

К сожалению, не со всеми трудностями удастся справиться так легко. Некоторые переменные становятся ненаблюдаемыми в момент генерации исключения. В частности, если в нашем примере вы выведете в окно наблюдения переменную *i*, то ее значение можно будет наблюдать при использовании описанных далее (см. раздел 2.6.7) точек прерывания и пошаговым выполнением приложения. Однако, в момент генерации исключения эта переменная становится недоступной и с этим ничего поделать невозможно.

Рассмотрим теперь работу с окном наблюдения. Перейдя в него, вы можете щелкнуть правой кнопкой мыши и во всплывшем меню выбрать ряд команд, в частности: Edit Watch (отредактировать наблюдаемое выражение) или Add Watch (вставить новое наблюдаемое выражение). В обоих случаях вы попадете в окно Watch Properties, представленное на рис. 2.35. Вы можете попасть в это окно из окна наблюдения и проще, нажав клавиши Ctrl-F5.

Рис. 2.35

Окно задания списка наблюдения Watch Properties



Рассмотрим коротко отдельные элементы окна Watch Properties. В окне редактирования Expression вы можете записать имя любой переменной или любое выражение, содержащее переменные, константы, функции. Окно редактирования Repeat count используется при наблюдении массивов и позволяет задать число наблюдаемых элементов массива. Например, если у вас в программе имеется массив **X**, вы можете просто указать в окне Expression имя переменной **X**. Тогда в окне наблюдений будут отображаться все элементы массива **X**. Но вы можете указать в окне Expression первый элемент массива **X[0]**, а в окне Repeat count написать, например, 10. Тогда в окне наблюдений будут отображаться только первые 10 элементов массива.

Окно редактирования Digits определяет число выводимых значащих разрядов чисел с плавающей запятой. Индикатор Enabled позволяет отключить вывод в окно наблюдения соответствующего выражения во время выполнения приложения. Это повышает производительность выполнения. А после того, как приложение остановлено и вам надо все-таки посмотреть данное выражение в окне наблюдения, выделите его в этом окне и сделайте на нем двойной щелчок. Откроется окно Watch Properties с загруженным в него выражением и вам останется только включить индикатор Enabled и щелкнуть OK.

Индикатор Allow Side Effects разрешает или запрещает отображение таких выражений, которые способны вызвать побочные эффекты. Например, вы можете записать в окне Expression выражение **++A**. Если индикатор Allow Side Effects выключен

(он выключен по умолчанию), то в окне наблюдений вы увидите рядом с выражением ++A текст: «Side effects are not allowed.» (побочные эффекты запрещены). Действительно, в данном случае, если отображать указанное выражение, то значение переменной A в программе изменится. Если же вы включите для этого выражения индикатор Allow Side Effects, то отображение соответствующего значения, на 1 большего A, будет производиться. Но учтите, что это будет изменять значение переменной A.

Радиокнопки в нижней части окна Watch Properties задают формат вывода значения переменной или выражения. По умолчанию включена кнопка Default. В этом случае формат определяется автоматически по типу отображаемого выражения. Но вы можете выбрать и другой формат. Например, вы можете воспользоваться этим радиокнопками, чтобы отображать некоторую целую переменную один раз в десятичном виде, а второй раз — в шестнадцатеричном виде.

Выпадающий список в окне редактирования Expression позволяет выбрать выражение из тех, которые использовались ранее, и при необходимости отредактировать его. Это удобно, если надо выводить в окно наблюдений похожие выражения. Например, если вам надо вывести значения **Form1->Label1->Caption**, **Form1->Label2->Caption** и **Form1->Label3->Caption**, то достаточно один раз написать это выражение, а в дальнейшем брать его из выпадающего списка и только менять в нем цифру.

В заключение посмотрим, как можно редактировать список выражений в окне наблюдения. Удалить какое-то выражение из этого списка можно просто выделив требуемое выражение и нажав Delete. Отредактировать ошибочное выражение можно, сделав на нем двойной щелчок, после чего вы окажетесь в окне Watch Properties и можете заниматься редактированием.

Вернемся к нашему примеру. Как видно из значений переменных (см. рис. 2.34), в момент появления ошибки выполнения значение A равно 10^{308} , а следующее значение должно быть равно 10^{312} . Такого значения переменная типа **double** хранить не может. Значит, или надо задать переменной A другой тип, или уменьшить скорость увеличения переменной. Теперь можно нажимать Ctrl-F2, прерывать выполнение и исправлять код. Но подождите это делать. Давайте посмотрим еще один инструмент отладки — окно оценки и модификации Evaluate/Modify.

2.6.5 Окно оценки и модификации Evaluate/Modify

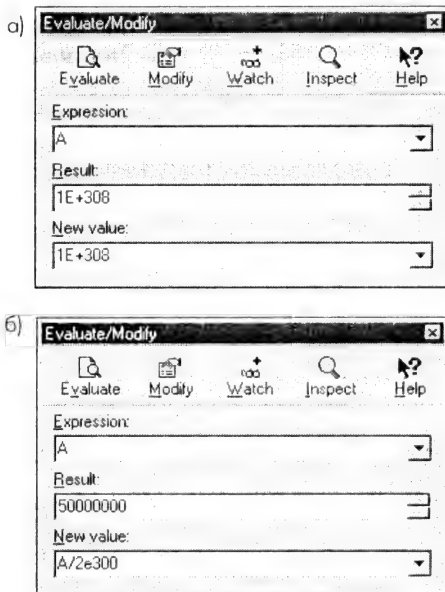
Окно оценки и модификации Evaluate/Modify позволяет вам в процессе отладки не только наблюдать, но и изменять значения переменных. Сделать это окно видимым можно командой Run | Evaluate/Modify. Соответствующую команду Debug | Evaluate/Modify можно также выбрать из контекстного меню, всплывающего при щелчке правой кнопкой в окне Редактора Кода.

Окно имеет вид, представленный на рис. 2.36 а. В его верхнем окошке редактирования Expression вы можете ввести имя переменной или выражение. После этого, щелкнув на кнопке Evaluate, вы увидите в окне Result значение этого выражения. Эти возможности пока ничем не отличаются от возможностей рассмотренного ранее окна наблюдений и даже слабее, поскольку дают значение только одной переменной или одного выражения. Но если вы указали в окошке Expression имя переменной, а не выражение, то вам становится доступной кнопка Modify, позволяющая изменить значение переменной. Т.е. вы можете вмешаться в процесс выполнения приложения и насильственно изменить значения переменных.

Пусть, например, мы решили изменить значение переменной A, вызвавшей переполнение, и посмотреть, как будут далее протекать события в приложении. Тогда мы должны в окошке Expression указать имя этой переменной, в окошке New value написать ее новое значение в виде числа или какого-то выражения (на рис. 2.36 б написано выражение «A/2e300») и нажать кнопку Modify. В результате

Рис. 2.36

Окно оценки и модификации в исходном состоянии (а) и после изменения значения А (б)



значение переменной в приложении изменится, что можно видеть в окошке Result. Измененное значение вы увидите и в окне наблюдения, если перейдете в него.

Поскольку вы резко уменьшили значение переменной А, можете после этого нажать F9, чтобы продолжить выполнение приложения. И оно будет продолжено, хотя, конечно, не надолго. После двух щелчков на кнопке ошибка переполнения возникнет вновь.

В данном примере исправление значения переменной, естественно, не имело практического смысла. Но в реальных приложениях это может оказаться очень полезным. Большое приложение может выполняться не доли секунды, как наш пример, а минуты, десятки минут и более. Тогда становится очень актуальной возможность оперативно исправить значения переменных, вызванные ошибкой, которую вы уже поняли, и продолжить отладку, не запуская выполнение опять с начала.

2.6.6 Выполнение приложения по шагам

Выше мы рассматривали способы собрать «статическую» информацию о состоянии приложения в момент, когда произошла ошибка выполнения. Но не всегда такая информация дает полную картину происходящего. Чаще для того, чтобы найти причину ошибки, надо выполнить какой-то фрагмент программы, наблюдая изменения переменных при выполнении каждой команды.

Для прохода фрагмента программы по шагам можно использовать команды:

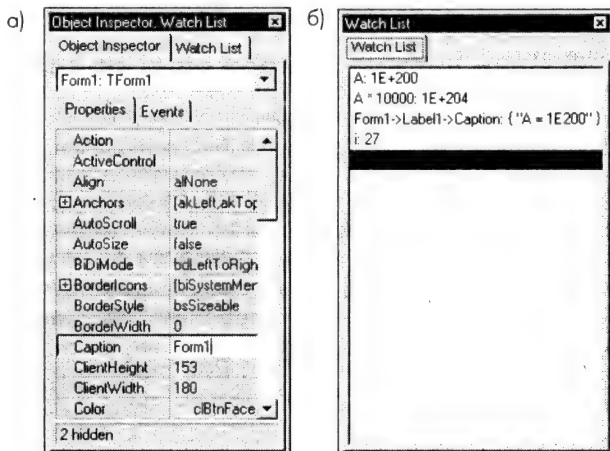
Команда	«Горячие» клавиши	Пояснения
Step Over (По шагам без захода в ...)	F8	Пошаговое выполнение строк программы, считая вызов функции за одну строку, т. е. вход в функции не производится.
Trace Into (Трассировка с заходом в ...)	F7	Пошаговое выполнение программы с заходом в вызываемые функции.

Команда	«Горячие» клавиши	Пояснения
Trace to Next Source Line (Трассировка до следующей строки)	Shift+F7	Переход к следующей исполняемой строке.
Run to Cursor (Выполнить до курсора)	F4	Команда выполняет программу до того выполняемого оператора, на котором расположен курсор в окне редактора кода.
Run Until Return (Выполнить до выхода из функции)	Shift+F8	Выполнение программы до выхода из текущей функции, останов на операторе, следующем за вызовом этой функции.
Show Execution Point (Показать точку выполнения)	-	Команда помещает курсор на операторе, который будет выполняться следующим.

Испытайте эти команды на нашем примере. Выведите значения интересующих вас переменных и выражений в окно наблюдения Watches. Это окно, с которым вы уже хорошо знакомы, является, как говорилось, встраиваемым. Этим удобно воспользоваться, встроив его, например, в Инспектор Объектов. В режиме проектирования окно наблюдения будет храниться на отдельной странице позади Инспектора Объектов (рис. 2.37 а), не занимая площадь экрана и нисколько не мешая работе. А во время выполнения приложения страницы Инспектора Объектов будут исчезать и при остановках вы можете наблюдать в окне Watches значения переменных (рис. 2.37 б). Для того, чтобы все это работало, надо сохранить описанную или любую другую конфигурацию отладочных окон с помощью команды View | Desktops | Save Desktop и командой View | Desktops | Set Debug Desktop задать эту конфигурацию как отладочную (подробности см. в разделе 2.2.9). Еще проще осуществлять подобные операции с конфигурациями соответствующими быстрыми кнопками вверху окна ИСР (см. таблицу 2.1 в разделе 2.2.3).

Рис. 2.37

Окно наблюдения, встроенное в Инспектор Объектов: режим проектирования (а) и режим отладки (б)



Выведя интересующие вас переменные в окно наблюдения и удобно разместив это окно, перейдите в ваш код и установите курсор на строке с оператором

```
A *= 10000;
```

Теперь нажмите F4, чтобы приложение выполнялось до тех пор, пока не дойдет до строки, в которой стоит ваш курсор. Приложение начнет выполняться. Нажмите в нем кнопку. Вы попадете в окно Редактора Кода, состояние которого будет таким, какое вы уже наблюдали ранее в случае ошибки (см. рис. 2.32): будет выделена строка, на которой стоял ваш курсор перед выполнением. Теперь вы можете, нажимая F7 или F8 (в данном случае это безразлично), выполнять операторы по шагам и в окне наблюдений видеть изменения переменных и выражений. Различие между F7 и F8 проявилось бы, если бы ваши операторы содержали вызов каких-то других функций, определенных в вашем модуле. В этом случае при нажатии F7 программа заходила бы внутрь этих вызываемых функций, а при нажатии F8 — не заходила бы.

Если вы прошли несколько циклов и вам это надоело, можете перевести курсор на оператор, следующий после цикла и задающий значение **Label1->Caption**. Нажмите F4. Тем самым вы сказали отладчику, что ему надо без остановок выполнять приложение до строки, указанной курсором. Все оставшиеся проходы цикла будут выполнены без остановок и программа остановится на указанной вами строке.

Если после этого вы нажмете клавишу F7 или F8, то результат будет различным. При нажатии F8 вы просто перейдете к следующему оператору — закрывающей фигурной скобке. А при нажатии F7 вы сначала попадете в заголовочный файл **dstring.h**, в котором объявлены функции работы со строками типа **AnsiString**, к которым вы неявно обращаетесь в операторе программы. И только после нескольких нажатий F7 вы вернетесь в свою программу.

2.6.7 Точки прерывания

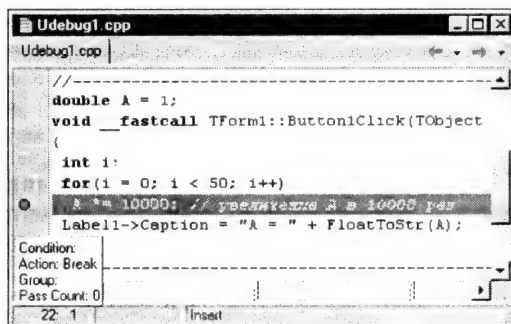
Выше мы рассмотрели останов приложения командой **Run to Cursor** — клавишей F4. Теперь рассмотрим более мощный инструмент — введение в приложение точек прерывания (breakpoint).

Чтобы ввести простую (безусловную) точку прерывания, достаточно в окне Редактора Кода щелкнуть мышью на полоске левее кода требуемой строки. Строка окрасится в красный цвет и на ней появится красная точка (рис. 2.38). Если вы теперь запустите приложение на выполнение и начнете с ним работать, то произойдет прерывание выполнения, как только управление перейдет к строке, в которой указана точка прерывания.

Такая точка прерывания дает тот же результат, что и описанное ранее выполнение до точки, указанной курсором, при нажатии клавиши F4. Но преимущество точек прерывания заключается в том, что их можно одновременно указать несколько в разных местах кода и в разных модулях. Приложение будет выполняться до тех пор, пока управление не перейдет к первой встретившейся в программе точке прерывания. Кроме того, как мы скоро увидим, в этих точках можно устанавливать условия прерывания.

Рис. 2.38

Окно Редактора Кода с выделенной точкой прерывания



Для того, чтобы убрать точку прерывания, достаточно щелкнуть мышью на красной точке левее кода соответствующей строки.

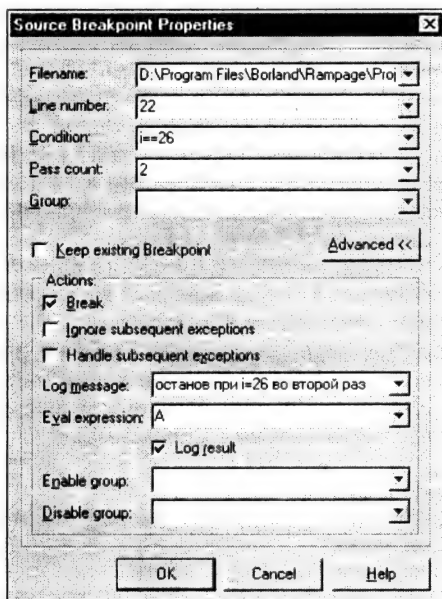
Точки прерывания можно устанавливать только на выполняемых операторах. Если вы, например, попытаетесь установить точку прерывания на строке, содержащей объявление переменной, то в момент запуска приложения в красной точке, выделяющей точку прерывания, появится крестик. Тем самым C++Builder предупреждает, что прерывания не будет, поскольку оператор невыполняемый. Аналогичный крестик может появиться и в том случае, если компилятор в процессе оптимизации кода убрал какой-то написанный вами оператор. Если вы все-таки хотите остановиться именно на этом операторе, вам надо отключить оптимизацию (см. раздел 14.2.9.3).

C++Builder дает возможность установить условия прерывания. Задержите курсор мыши над красной точкой слева от строки, где вы установили прерывание. Вы увидите (рис. 2.38) всплывший ярлычок с характеристиками данной точки прерывания. По умолчанию никаких условий останова не задается. Поэтому до сих пор останова были при каждой передаче управления на точку прерывания. Попробуем задать условия прерывания. Пусть, например, мы хотим остановиться на точке прерывания, которую мы уже ввели, но только на 26-м цикле, причем только при повторном проходе цикла (повторном щелчке пользователя на кнопке нашего приложения). Иначе говоря, мы хотим остановиться, когда переменная *i* будет иметь значение 26, причем не в первый, а во второй раз.

Щелкните правой кнопкой мыши на красной точке в строке, где вы ввели прерывание (рис. 2.38). Из всплывшего меню выберите раздел Breakpoint properties. Перед вами откроется окно, представленное на рис. 2.39. Это окно в C++Builder 5 существенно богаче по своим возможностям, чем аналогичное окно в C++Builder 4.

Рис. 2.39

Окно (в развернутом виде) задания свойств прерывания в указанной строке файла в C++Builder 5



Два верхних окошка редактирования — Filename (имя файла) и Line Number (номер его строки) в данном режиме недоступны. Они автоматически заполнились в момент, когда вы установили в своем коде точку прерывания.

Окошко Condition (условие) позволяет вам ввести некоторое условное выражение. Прерывание будет происходить только в случае, если значение этого выражения равно **true**. В нашем примере вы можете указать условие **останова «i=26»**.

Окошко Pass Count позволяет указать, при котором по счету выполнении записанного условия произойдет останов. В нашем примере нас интересует второй проход цикла. Поэтому задайте в окошке Pass Count значение 2.

Окошко Group (группа) позволяет задать имя группы, к которой относится данное прерывание. C++Builder 5 позволяет отнести прерывание к какой-то группе. Группировка прерываний позволяет сделать целиком ту или иную группу доступной или недоступной. Это делается с помощью показанных внизу на рис. 2.39 окошек **Enable Group** (доступная группа) и **Disable Group** (недоступная группа). В этих окошках имя группы выбирается из выпадающих списков, содержащих имена введенных ранее групп.

Кнопка **Advanced** разворачивает нижнюю часть окна задания свойств прерывания. В ней вы можете указать некоторые действия (Action), выполняемые в момент прерывания. Индикатор **Break**, установленный по умолчанию, обеспечивает стандартную реакцию — останов выполнения приложения. Группа индикаторов **Ignore subsequent exception** (игнорировать последующее исключение) и **Handle subsequent exception** (обрабатывать последующее исключение), из которых установить можно только один, определяют появление при генерации очередного исключения сообщения отладчика (см. раздел 2.6.3, рис. 2.31). Если установлен индикатор **Ignore subsequent exception**, то это сообщение не появляется. Если установлен индикатор **Handle subsequent exception**, то это сообщение появляется. Если не установлен ни один индикатор, то все определяется настройками отладчика C++Builder на странице **Language Exceptions** окна настройки отладчика (см. раздел 14.2.8).

Окошко **Log Message** позволяет записать некоторое сообщение, которое будет появляться в момент прерывания в окне протокола событий **Event Log** (см. раздел 2.6.9). Окошко **Eval Expression** позволяет записать некоторое выражение, которое будет вычисляться при прерывании. При установленном индикаторе **Log Result** это выражение и результат его вычисления вы сможете увидеть в том же окне **Event Log**.

С учетом возможностей задания сообщений и выражений, появляющихся в окне **Event Log**, вам, вероятно, становится более понятным наличие индикатора **Break**. Если вы выключите этот индикатор и включите индикатор **Ignore subsequent exception**, то никаких остановов выполнения приложения в точке прерывания или при генерации исключения не будет происходить. Но завершив выполнение приложения и посмотрев протокол **Event Log**, вы увидите там соответствующие сообщения, которые позволят вам отследить логику работы приложения, понять, какие фрагменты кода и сколько раз выполнялись.

Внесите в окно задания свойств прерывания данные, показанные на рис. 2.39, щелкните на **OK** и запустите выполнение приложения. После первого нажатия вами кнопки приложения останова не произойдет, а после второго нажатия выполнение остановится. Вы сможете увидеть в окне наблюдений, что в этот момент **i = 26**. Затем, при желании, вы можете пройти по шагам (с помощью клавиши **F7**) последний цикл перед появлением ошибки. А если в окне рис. 2.39 вы выключите индикатор **Break**, то останова в точке прерывания не будет, а после генерации исключения вы сможете выполнить команду **View | Debug Windows | Event Log**, и в окне протокола событий найдете, в частности, строки

```
Breakpoint Message: останов при i=26 во второй раз Process Pdebug1.exe (0xFFFFAF9B9)
```

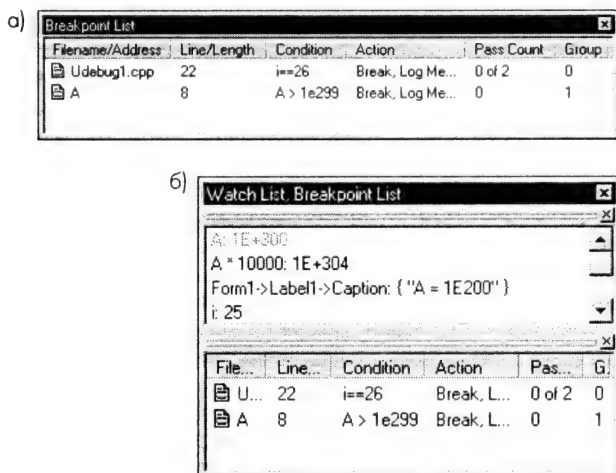
```
Breakpoint Expression A: 1E+304 Process Pdebug1.exe (0xFFFFAF9B9)
```

Это введенное вами сообщение («останов при i=26 во второй раз») и выражение (A) с посчитанным результатом.

Список введенных вами точек прерывания хранится в списке, который вы можете в любой момент (и до выполнения программы, и при останове) увидеть, если выполните команду View | Debug Windows | Breakpoints. Откроется окно Breakpoint List (рис. 2.40 а). В нем вы увидите введенную вами ранее точку прерывания (на рис. 2.40 видна еще одна точка, которую мы введем позднее). Окно точек прерывания удобно встраивать в окно наблюдения (см. рис. 2.40 б) и фиксировать такую конфигурацию окон как отладочную. Тогда в моменты остановки вы будете видеть и наблюдаемые величины, и информацию о точках прерывания.

Рис. 2.40

Окно точек прерывания в виде отдельного окна (а) и встроенное в окно наблюдения (б)



Условные точки прерывания — мощный инструмент отладки приложений, содержащих циклы. В нашем примере мы знаем, что ошибка происходит именно при втором проходе цикла, поскольку каждый проход мы сами запускаем нажатием кнопки приложения. Но если бы в более сложной программе мы этого не знали, то условная точка прерывания позволила бы нам определить, на каком именно проходе произошла ошибка. Для этого можно в окне задания свойств точки прерывания указать большое значение Pass Count, например, 50. Тогда, конечно, прерывание в этой точке не произойдет, поскольку раньше будет зафиксирована ошибка. Но если после прерывания вследствие этой ошибки вы посмотрите окно Breakpoint List, то увидите, что там указано, сколько произошло проходов через точку прерывания с выполнением заданного условия.

Теперь обсудим некоторые возможности окна точек прерывания. Щелкните в окне правой кнопкой мыши и во всплывшем меню выберите команду Add — добавить точку прерывания. Вам будет предложено три варианта: Source Breakpoint — точка прерывания в файле приложения, Address Breakpoint — прерывание при переходе управления по заданному адресу, и Data Breakpoint — прерывание при изменении данных по заданному адресу. Правда, если вы работаете в окне Breakpoint List не во время выполнения вашего приложения, две последние команды будут недоступны. Это связано с тем, что пока приложение не выполняется, адреса команд и переменных еще не известны.

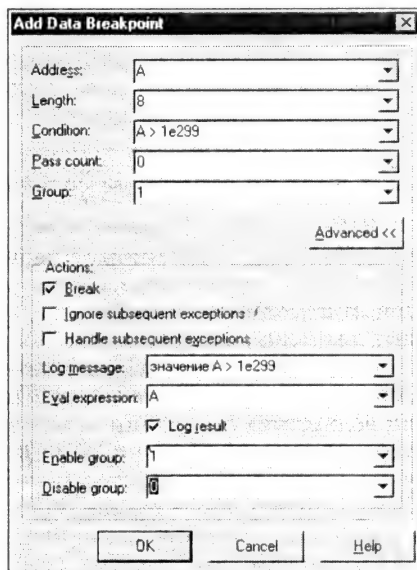
Давайте рассмотрим пример использования прерывания при изменении данных по заданному адресу. В нашем приложении мы знаем единственный оператор, который изменяет значение переменной А. Но в сложном приложении подобных операторов может быть много. И тогда мы не можем знать, какой из них увеличивает значение А настолько, что дальнейшее выполнение приложения грозит переполнением, или какой оператор заносит в А недопустимое значение. В подобных случаях помогает установка прерывания при изменении данных по заданному ад-

ресу — в нашем примере по адресу, отведенному для **A**. Вы не устанавливаете, на какой строке кода хотите прервать выполнение. Прерывание произойдет на том операторе, который занесет в указанную вами переменную некоторое значение, если будут выполнены указанные вами условия.

Пусть мы хотим остановиться на операторе, заносащем новое значение в адрес **A**, причем при значении $A > 10^{299}$. Поскольку отладчик не знает адреса переменных, пока приложение не загружено в память, то прежде, чем задавать точку прерывания, в данном случае надо запустить приложение на выполнение. Затем, не закрывая приложения и ничего с ним не делая, вернитесь в ИСП C++Builder. В окне Breakpoint List щелкните правой кнопкой мыши и во всплывшем меню выберите команду Add. Из предложенных вам трех вариантов точек прерывания выберите Data Breakpoint — прерывание при изменении данных по заданному адресу. Откроется окно (рис. 2.41), подобное рассмотренному ранее (см. рис. 2.39). Только здесь в окошке редактирования Address задается переменная, при изменении которой наступает прерывание. Окошко длины переменной (Length) обычно заполняется автоматически. Окна Condition и Pass Count имеют тот же смысл, что в окне рис. 2.39.

Рис. 2.41

Окно (в развернутом виде) задания свойств прерывания при изменении данных по заданному адресу



Заполнив окно так, как показано на рис. 2.41, щелкните на ОК. Вы вернетесь в окно списка точек прерывания, в котором появится введенная вами новая точка (рис. 2.40 а).

Чтобы убедиться в работе новой точки прерывания, отключите прежнюю. Для этого выделите ее в окне списка точек прерывания, щелкните на ней правой кнопкой мыши и из контекстного меню выберите команду Properties. В появившемся диалоговом окне (рис. 2.39) введите имя группы (например, «0») и выберите это же имя в окошке Disable Group. Тем самым вы сделаете недоступной прежнюю точку прерывания. Можно поступить проще: щелкнуть правой кнопкой мыши на строке прежней точки прерывания и во всплывшем меню снять установку индикатора Enabled. На всякий случай проверьте, что в новой точке прерывания индикатор Enabled включен. Теперь вернитесь в ваше выполняющееся приложение и щелкните кнопкой. Приложение будет выполняться, но заметно медленнее, чем раньше. Это влияние накладных расходов на проверку условий при каждом изменении **A**. После второго щелчка приложение остановится на операторе

```
A:=A*10000;
```


при состоянии приложения, когда $A = 10^{300}$ (см. рис. 2.40 б). Т.е. прерывание, как и было заказано, произошло после того, как соответствующим образом изменилась переменная **A**.

Учтите, что доступность точки прерывания по изменению переменной сохраняется только в течение данного сеанса выполнения приложения. По завершении выполнения точка становится недоступной и при следующем запуске ее надо повторно активизировать, устанавливая с помощью контекстного меню ее индикатор **Enabled**.

Помимо описанных команд, задающих точки прерывания в C++Builder 5, имеется еще несколько команд, осуществляющих те же функции. Это, прежде всего, команда Run | Add Breakpoint (добавить точку прерывания) с разделами Source Breakpoint, Address Breakpoint, Data Breakpoint и Module Load Breakpoint. Первые три раздела выполняют те же функции, что и одноименные им разделы контекстного меню списка точек прерывания, рассмотренные ранее. А последний раздел позволяет задать прерывание при загрузке в память указанного модуля (обычно библиотеки **.dll** или пакета **.bpl**). Имеется также возможность задать прерывание по изменению переменной из всплывающего меню окна наблюдения (команда Break When Changed). Соответствующая точка возникает в списке точек прерывания, но доступна только в течение данного сеанса выполнения приложения. По завершении выполнения точка становится недоступной и при следующем запуске ее надо повторно активизировать, устанавливая с помощью контекстного меню ее индикатор **Enabled**.

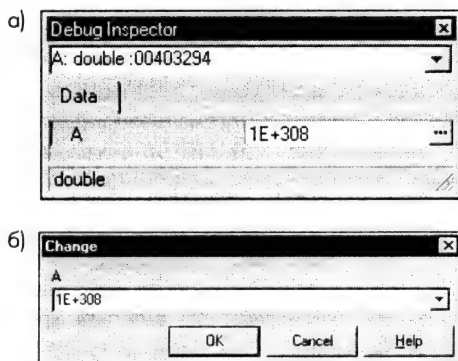
2.6.8 Использование окна Инспектора Отладки Debug Inspector

В C++Builder 5 имеется еще одно средство отладки — Инспектор Отладки Debug Inspector. Инспектор Отладки позволяет вам получить исчерпывающую информацию о любой переменной в приложении и дает возможность, как и окно оценки и модификации Evaluate/Modify, изменить значение переменной и продолжить выполнение приложения с этим новым значением.

Вызов этого инструмента осуществляется командой Run | Inspect, которая доступна только во время выполнения приложения при останове средствами отладки или вследствие генерации исключения. При останове вы можете поставить курсор в окне Редактора Кода на имя интересующей вас переменной и выполнить команду Run | Inspect. Другой способ — вызвать ту же команду из всплывающего меню (Debug | Inspect). Ну, а проще всего — нажать «горячие» клавиши Alt+F5. Попробуйте сделать это при генерации исключения в нашем тестовом приложении. Если после прерывания выполнения вы поставите курсор на пустое место в коде и вызовите Инспектор Отладки, перед вами откроется окно, в котором вы можете написать имя интересующей вас переменной, например, **A** и щелкнуть OK. Вы увидите окно, представленное на рис. 2.42 а.

Рис. 2.42

Окна Инспектора Отладки: наблюдения (а) и изменения (б) значения переменной

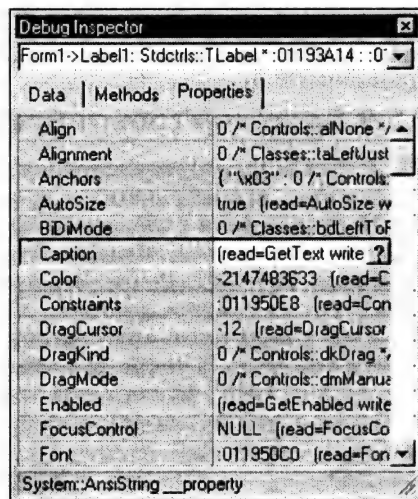


В окне содержатся сведения об указанной вами переменной. Нажав кнопку с многоточием, вы можете увидеть следующее окно Инспектора Отладки — окно изменения *Change*, представленное на рис. 2.42 б. В этом окне вы можете изменить значение переменной и оно изменится в выполняемой программе, так что при продолжении выполнения приложения оно будет выполняться с заданным вами значением переменной.

Инспектор Отладки позволяет исследовать различные данные: переменные, массивы, классы, функции, указатели. На рис. 2.43 приведено окно, которое вы могли бы увидеть, если бы в качестве объекта исследования указали кнопку **Form1->Label1**. Как видите, это окно имеет три страницы, из которых одна — страница свойств *Properties*, показана на рисунке. На этой странице вы можете увидеть перечисление всех свойств компонента, их значения и функции их чтения и записи.

Рис. 2.43

Окно Инспектора Отладки для объекта *Label1*



Если вы хотите изменить какое-то свойство (конечно, не из тех, которые только для чтения), вы можете выделить это свойство и нажать появившуюся около него кнопку с многоточием. Появится окно изменения *Change*, аналогичное приведенному на рис. 2.42 б, в котором вы можете ввести новое значение свойства.

Не все значения свойств могут быть в момент останова досчитаны до конца. В этом случае, если вы выделите курсором это свойство, около него появляется кнопка со знаком «?». Она видна на рис. 2.43 в строке свойства **Caption**. Из рисунка видно, что значение надписи метки не посчитано и не выведено в окне. Если вы нажмете кнопку со знаком «?», то значение будет досчитано и, пока эта кнопка нажата, при каждом останове выполнения свойство будет досчитываться до конца.

Находясь в окне Инспектора Отладки, можно щелкнуть правой кнопкой мыши и выбрать одну из следующих команд:

Range	Просмотр данных в заданном диапазоне
Change	Перейти в окно <i>Change</i> для изменения значения элемента
Show Inherited	Если этот флаг включен, то на страницах окна отображаются все свойства и методы, как объявленные в данном классе, так и наследуемые. Если флаг выключен, то отображается только то, что объявлено в данном классе
Show Fully Qualified Names	Отображение наследуемых элементов с их полными именами

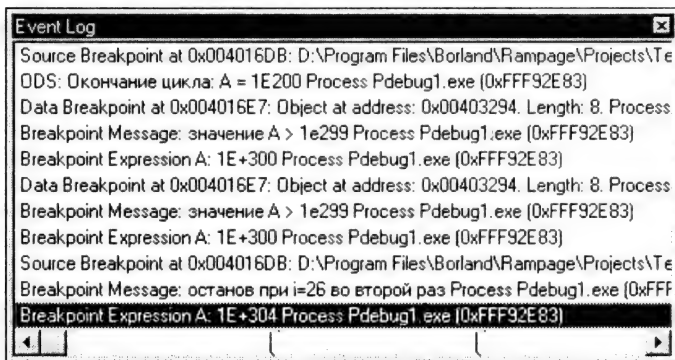
Inspect	Открывает новое окно для выделенного вами элемента данных. Это позволяет детальнее исследовать данные типа структур, классов, массивов и т.п. и только для таких данных этот раздел доступен
Descend	Аналогична команде Inspect , но детализирующие данные появляются не в отдельном, а в том же самом окне. В дальнейшем можно вернуться в исходное окно, воспользовавшись расположенным вверху окна выпадающим списком, в котором накапливаются просмотренные объекты
New Expression	Эта команда позволяет вам задать новое выражение для анализа
Type Cast	Позволяет вам указать другой тип для рассматриваемого объекта, например, указать тип нетипизированного указателя

2.6.9 Протокол событий, функция OutputDebugString

В C++Builder предусмотрена возможность просматривать протокол сообщений о событиях, происходящих в процессе выполнения приложения в режиме отладки. Протокол этих сообщений вы можете посмотреть в процессе выполнения или после его окончания, выполнив команду View | Debug Windows | Event Log или нажав клавиши Ctrl-Alt-E. В открывшемся окне Event Log (его пример приведен на рис. 2.44) вы увидите протокол событий. На рис. 2.44 вы можете видеть, в частности, сообщения точек прерывания, введенных ранее в разделе 2.6.7: три последние строки относятся к точке прерывания по $i = 26$, строки с 3 по 5 — к точке прерывания по $A > 10^{299}$. Щелкнув в окне правой кнопкой мыши, вы можете сохранить протокол в файле, прокомментировать его, очистить.

Рис. 2.44

Окно Event Log



Какие именно сообщения отображаются в этом окне определяется настройкой отладчика, которая рассмотрена в разделе 14.2.8. Для пользователей, не слишком сведущих в системном программировании, можно рекомендовать ограничиться сообщениями о точках прерывания и сообщениями, генерируемыми функцией **OutputDebugString**. Об этой функции надо сказать особо. Это функция API Windows, определенная следующим образом:

```
VOID OutputDebugString(LPCTSTR lpOutputString);
```

Ее параметр **lpOutputString** является указателем на строку текста с нулевым символом в конце.

Функция **OutputDebugString** в процессе отладки выдает сообщение, которое вы можете наблюдать в окне протокола сообщений о событиях Event Log. Но если

отладчик отключен или если выполняемый модуль вашего приложения запускается непосредственно, а не из среды C++Builder, то функция **OutputDebugString** ничего не делает. Таким образом, вы можете внести в разных местах своего приложения вызовы **OutputDebugString** с соответствующими сообщениями, которые покажут вам ход выполнения приложения в режиме отладки. А когда вы или другие пользователи впоследствии запустят приложение в обычном режиме, наличие в нем вызовов ничем не помешает, кроме очень незначительных затрат времени и незначительного увеличения объема модуля.

Можете опробовать этот инструмент в тестовом приложении, рассмотренном ранее, вставив, например, в конце функции **TForm1::Button1Click** оператор

```
OutputDebugString(("Окончание цикла: A = " + FloatToStr(A)).c_str());
```

Результат выполнения тестового приложения с этим оператором и приведен на рис. 2.44 (см. вторую строку окна). Только прежде, чем запускать ваше приложение, посмотрите в разделе 14.2.8, как отключить системные сообщения. В противном случае вы с трудом найдете свои сообщения среди множества системных.

2.6.10 Другие средства отладки

В C++Builder имеется еще немало инструментария, позволяющего проводить отладку на более детальном уровне. Это, прежде всего, окно CPU (команда View | Debug Windows | CPU), позволяющее отследить ход выполнения проекта на уровне команд макроассемблера. Окно стека (команда View | Debug Windows | Call Stack) позволяет определить последовательность вызванных функций, не только тех, которые вызываются явно вашим приложением, но и всех неявно вызываемых функций библиотек. Окно модулей (команда View | Debug Windows | Modules) показывает список всех модулей, загруженных в память при выполнении вашего приложения. Это окно позволяет, в частности, если выделить на его левой верхней панели имя вашего модуля **.exe**, увидеть в левой нижней панели список использованных заголовочных файлов. Двойной щелчок на строке соответствующего файла загрузит его в Редактор Кода и вы сможете попробовать понять, что именно из данного файла использует ваше приложение и в чем суть возникших проблем.

Эти и еще некоторые другие средства отладки предназначены для пользователей, чувствующих в себе силы погрузиться в тонкости работы приложения на уровне макроассемблера. Подавляющему большинству пользователей это вряд ли интересно и нужно. Поэтому в рамках данной книги мы не будем погружаться в эти глубины.

2.6.11 Некоторые приемы программирования, встраивающие отладку в код

Материал данного раздела предполагает наличие у читателя определенных сведений о программировании на языке C++ и о некоторых компонентах C++Builder. Тем, кто этими сведениями пока не обладает, можно пропустить этот раздел и вернуться к нему позднее.

В C++Builder имеется полезный инструмент для отладки — функция **assert**. Ее синтаксис:

```
#include <assert.h>
void assert(int test);
```

Функция применяется при отладке программ для проверки истинности утверждений **test**, которые по замыслу должны быть истинны, но в силу каких-то ошибок могут нарушаться. Если проверяемое утверждение **test** окажется ложным (возвращает 0), работа приложения завершается вызовом функции **abort** и выдается сообщение об ошибке.

Попробуйте использовать эту процедуру в вашем тестовом приложении. Вставьте директиву препроцессора

```
#include <assert.h>
```

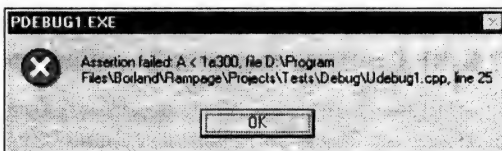
и измените, например, цикл в приложении на следующий:

```
for(i = 0; i < 50; i++)
{
    A *= 10000;
    assert(A < 1e300);
}
```

В этом коде проверяется условие $A < 10^{300}$ и если оно нарушается, то выдается сообщение об ошибке. Если вы запустите свое приложение, то при повторном щелчке на кнопке получите сообщение, представленное на рис. 2.45. В нем, отображается условие, которое проверялось, а также модуль и строка, в которой произошло данное событие.

Рис. 2.45

Окно сообщения функции assert



Закономерен вопрос: «А зачем нужна специальная функция **assert**, если то же самое можно было бы сделать простым оператором **if**?». Особенностью и преимуществом функции **assert** является то, что она реализована в виде макроса, который разветвляется в соответствующую функцию только в случае, если не определен идентификатор **NDEBUG**. Поэтому достаточно вставить в ваш файл перед директивой

```
#include <assert.h>
```

директиву

```
#define NDEBUG
```

и откомпилировать файл, как вызов функций, в которые разворачиваются макросы **assert**, исчезнут. Таким образом вы можете одной строкой отключить множество отладок, которые разбросаны по разным фрагментам вашей программы. При этом отладки не просто перестанут работать, но они исчезнут из результирующего выполняемого модуля, так что не будут увеличивать его размер и потреблять вычислительные ресурсы.

Впрочем, возможности **assert** слишком скромны, чтобы дать достаточную для отладки информацию. Вы легко можете сами определить свой макрос с более широкими возможностями, позволяющий включать любые переданные в него сообщения. В качестве примера ниже приведен текст макроса **MyWarning**, который построен по принципу **assert**, но позволяет отобразить дополнительное переданное в него сообщение **msg**. Только обратите внимание на то, что директива, следующая после строки комментария, должна быть записана в тексте вашего модуля в одну строку, без переносов. В приведенном тексте переносы введены просто из-за того, что при печати книги она не может уместиться в одну строку.

```
#ifdef NDEBUG
#define MyWarning(p,msg) ((void)0)
#else
// следующая директива должна быть записана в одну строку
#define MyWarning(p,msg) ((p) ?(void)0 : (ShowMessage(msg)+
    ", (" +AnsiString(p)+")\nфайл " + AnsiString(__FILE__) +
    ", строка " + IntToStr(__LINE__)))
#endif
```

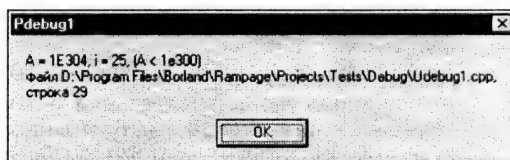
Чтобы воспользоваться этим макросом в нашем примере, можете вставить в цикл оператор

```
MyWarning(A < 1e300, "A = " + FloatToStr(A) + ", i = " +
        IntToStr(i));
```

В момент, когда значение переменной **A**, превысит 10^{300} , этот макрос выдаст сообщение вида, показанного на рис. 2.46.

Рис. 2.46

Сообщение вашего макроса MyWarning



Если вы хотите не просто отобразить сообщение, но и завершить программу, проще объявить некоторую функцию, в вызов которой разворачивать макрос. В качестве примера ниже приведен макрос и соответствующая функция, обеспечивающие примерно те же возможности, что и рассмотренный выше макрос, но завершающие работу приложения.

```
#ifdef NDEBUG
#define MyAssert(p,msg) ((void)0)
#else
// следующая директива должна быть записана в одну строку
#define MyAssert(p,msg) ((p) ?(void)0 :
                        (FMyAssert((msg), __FILE__, __LINE__)))
#endif
```

```
void FMyAssert(String msg, String sFile, int Line)
{
    ShowMessage((msg)+"\nфайл " + sFile + ", строка " +
        IntToStr(Line));
    abort();
}
```

Вызов такого макроса **MyAssert** может выглядеть аналогично вызову предыдущего макроса:

```
MyAssert(A < 1e300, "A = " + FloatToStr(A) + ", i = " + IntToStr(i));
```

Преимущество реализации отладок макросами заключается в том, что вы можете вставить их вызовы, которые выглядят весьма компактно, в различных частях своей программы. А когда все отладки закончены, можете отключить их все, введя в свой файл директиву

```
#define NDEBUG
```

Вы можете, конечно, обойтись и без макросов, введя в тех или иных местах программы операторы, обеспечивающие отладку. При этом целесообразно окружать эти операторы отладки следующими директивами:

```
#ifndef NDEBUG
    операторы отладки
#endif
```

Операторы отладки должны обеспечивать отладочную печать, которая помогает следить за ходом выполнения приложения. В C++Builder для отображения отладочной информации, на мой взгляд, удобен компонент **TMemo** — многострочное окно редактирования. В него можно заносить отладочные данные и тут же, во время выполнения приложения просматривать их. Чтобы этот компонент не занимал место на форме, его можно выводить в отдельное окно. Приведу без особых ком-

ментариев, как это можно было бы сделать в нашем тестовом примере, если бы нам вдруг для отладки захотелось наблюдать, как изменяются в цикле переменные.

```
// форма отладочного окна
TForm * FDebug;
//компонент вывода в отладочном окне
TMemo * Memol;
...
//-----
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    #ifndef NDEBUG
    // создание отладочного окна и окна редактирования в нем
    FDebug = new TForm(Application);
    FDebug->Caption = "Отладка";
    Memol = new TMemo(FDebug);
    Memol->Parent = FDebug;
    Memol->ScrollBars = ssVertical;
    Memol->Align = alClient;
    FDebug->Show();
    #endif
}
//-----
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
    #ifndef NDEBUG
    // освобождение памяти
    FDebug->Release();
    #endif
}
//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    for(int i = 0; i < 50; i++)
    {
        A *= 10000;
    }
    #ifndef NDEBUG
    //отладочная печать
    Memol->Lines->Add("A = "+FloatToStr(A)+"", i="+IntToStr(i));
    #endif
}
Label1->Caption = "A = " + FloatToStr(A);
}
```

В этом коде прокомментировано все, относящееся к отладке. В обработчике события формы **OnCreate** создается окно отладки **FDebug** и в нем — окно редактирования **Memol**. В цикл нашего примера вставлен вывод информации в это окно. В событии формы **OnDestroy** предусмотрено уничтожение отладочного окна. Причем, все это делается только если не определен идентификатор **NDEBUG**.

Попробуйте сделать в приложении указанные изменения и запустить его на выполнение. Вы увидите в процессе выполнения окно, показанное на рис. 2.47, в которое по мере работы приложения будут заноситься данные. Если бы это было какое-то сложное приложение, то вы непосредственно в процессе выполнения могли бы просматривать эти данные и наблюдать за ходом работы.

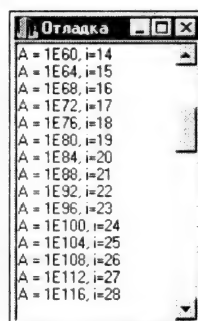
После того, как вы отладили приложение, вы можете легко удалить всю отладочную печать, введя директиву

```
#define NDEBUG
```

Здесь, впрочем, надо сделать одно замечание. Сложное приложение, увы, не может быть сделано без ошибок. Ошибки могут проявляться иногда через месяцы и годы работы у пользователя в каких-то экзотических ситуациях, при каких-то

Рис. 2.47

Окно отладки



редких сочетаниях исходных данных или каких-то непредусмотренных действиях пользователя. Причем ошибки могут возникнуть на компьютере пользователя, где, естественно, нет ваших исходных файлов а, возможно, нет и C++Builder. В таких ситуациях могла бы помочь отладка проекта непосредственно на компьютере пользователя. Но при рассмотренных ранее подходах отладка в завершенном выполняемом модуле отсутствует.

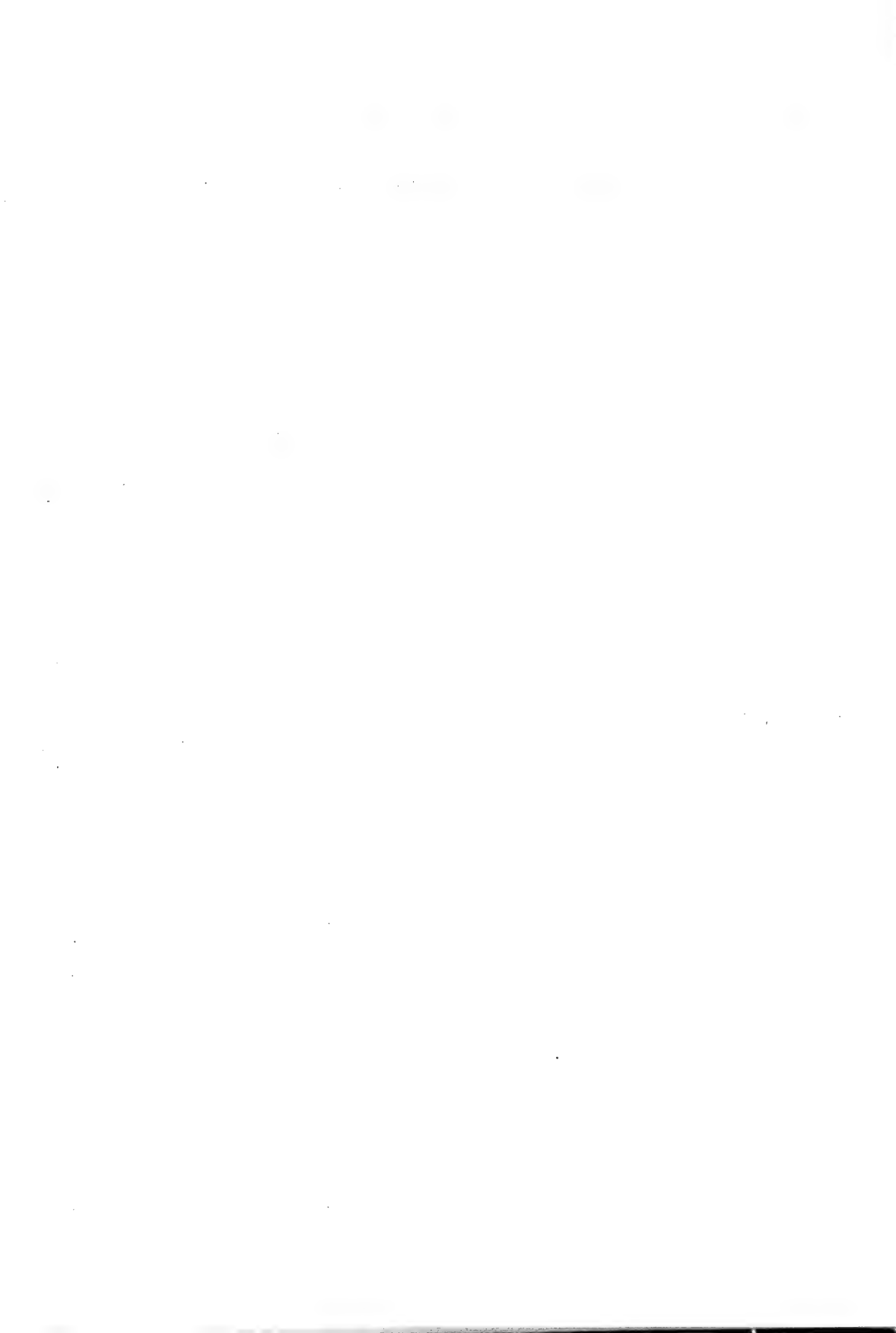
Исходя из этого я, например, предпочитаю в готовом приложении оставить некоторые возможности отладочной печати. Включать ее можно, например, созданием в текущем каталоге текстового файла с условным именем **Debug.txt**. Тогда можно ввести в проект некоторую переменную **debug**, значение которой определяется наличием или отсутствием этого файла:

```
bool debug = FileExists("Debug.txt");
```

Тогда те отладки (конечно, немногочисленные), которые желательно оставить в приложении, могут выполняться по условному оператору

```
if (debug)
...
```

При нормальной работе, когда файла **Debug.txt** нет, никаких отладок не будет и пользователь даже не будет подозревать об их наличии в приложении. Но стоит любым текстовым редактором создать такой файл, и вы получаете возможность наблюдать за ходом выполнения приложения. Конечно, реально лучше этот механизм несколько усложнить, заноса в файл **Debug.txt**, например, тот или иной уровень отладки, который может читаться приложением. Тогда, ничего не изменяя в самом приложении, можно управлять отладкой, делая ее более или менее подробной.



Обзор компонентов библиотеки C++Builder

3.1 Страницы палитры компонентов

Палитра компонентов VCL — библиотеки визуальных компонентов C++Builder, имеет ряд страниц, на которых скомпонованы пиктограммы всех компонентов, предопределенных в C++Builder. Вы можете изменять компоновку страниц (см. раздел 14.2.2), а также добавлять новые страницы, вносить на страницы свои новые компоненты и шаблоны компонентов (см. главу 7).

Предопределенные в C++Builder страницы палитры зависят от версии, с которой вы работаете. По умолчанию в палитре C++Builder 5 имеются страницы:

Standard	Стандартная, содержащая наиболее часто используемые компоненты
Additional	Дополнительная, являющаяся дополнением стандартной
Win32	32-битные компоненты в стиле Windows 95/98/2000 и NT
System	Системная, содержащая такие компоненты, как таймеры, плееры и ряд других
Data Access	Доступ к данным через Borland Database Engine (BDE)
Data Controls	Управление данными
ADO	Связь с базами данных через Active Data Objects (ADO) — множество компонентов ActiveX, использующих для доступа к информации баз данных Microsoft OLEDB (только начиная с C++Builder 5)
Interbase	Прямая связь с Interbase, минуя Borland Database Engine (BDE) и Active Data Objects (ADO) (только начиная с C++Builder 5)
Midas	Построение приложений баз данных с параллельными потоками (только в вариантах Client/Server и Enterprise и только начиная с C++Builder 4)
InternetExpress	Построение приложений InternetExpress — одновременно приложений сервера Web и клиента баз данных с параллельными потоками (только начиная с C++Builder 5)
Internet	Компоненты для приложений, работающих с Интернет
FastNet	Различные протоколы доступа к Интернет (только начиная с C++Builder 5)
Decision Cube	Многомерный анализ данных (только в вариантах Client/Server и Enterprise)
Qreport	Быстрая подготовка отчетов
Dialogs	Системные диалоги типа «Открыть файл» и др.

Win 3.1	Windows 3.x, компоненты в стиле Windows 3.x (оставлены для обратной совместимости)
Servers	Оболочки VCL для распространенных серверов COM (только начиная с C++Builder 5)

Имеются еще две страницы, содержащие примеры:

Samples	Образцы: различные интересные, но не до конца документированные компоненты
ActiveX	Примеры активных элементов ActiveX

Примеры на страницах Samples и ActiveX не документированы в C++Builder и во встроенной справке сведения о них отсутствуют. Однако, исходные тексты примеров со страницы Samples имеются в каталоге ...\\EXAMPLES\\CONTROLS\\SOURCE. Вы можете их просмотреть и понять, как построены эти примеры и как ими пользоваться.

Примеры со страницы ActiveX также не документированы. Но если вы перенесете соответствующий компонент на форму и щелкнете на нем правой кнопкой мыши, то во всплывшем меню можете выбрать команду Property и некоторые другие, которые отобразят диалоговые окна, помогающие задать необходимые свойства компонента.

Многие из компонентов страниц Samples и ActiveX надо рассматривать скорее именно как примеры создания компонентов. Их полезно изучить, но для практического использования в приложениях многие из них не очень приспособлены.

Компоненты страницы Win 3.1 не рекомендуется использовать в 32-разрядных версиях Windows. Они предназначены для приложений, которые должны функционировать в любой версии Windows, включая Windows 3.x. Для 32-разрядных приложений имеются аналоги большинства компонентов страницы Win 3.1, которые приведены в таблице 3.1.

Таблица 3.1. Соответствие компонентов страницы Win 3.1 и новых 32-разрядных компонентов

Компонент Win 3.1	Новый компонент	Страница нового компонента
DBLookupList	DBLookupListBox	Data Controls
DBLookupCombo	DBLookupComboBox	Data Controls
TabSet	TabControl	Win32
Outline	TreeView	Win32
Header	HeaderControl	Win32
NoteBook	PageControl	Win32
TabbedNoteBook	PageControl	Win32

В различных главах данной книги рассмотрено большинство компонентов библиотеки C++Builder. Исключение составляют страницы Internet, MIDAS, InternetExpress и FastNet, которые не рассматриваются просто из-за ограничения на объем книги.

Все компоненты страниц Data Access, Data Controls, ADO, Interbase, Decision Cube рассмотрены подробно в главах 9, 10, 11 и в данной главе не обсуждаются. Компоненты страницы QReport рассматриваются в главе 11 в разделе 11.2. Компоненты мультимедиа рассматриваются в главе 5 (раздел 5.2). Компоненты страницы Servers рассматриваются в главе 6 раздел 6.4.4 и в главе 11 раздел 11.3

В справочной части книги в главе 14 приведен полный состав страниц палитры компонентов. Но в данном разделе мы будем рассматривать компоненты, komponуя их не по страницам, а по выполняемым функциям. При этом ограничимся только самым общим описанием и сравнением компонентов общего назначения, которые применяются в большинстве приложений. Более подробные сведения об этих и других компонентах вы почерпнете в последующих главах.




К сожалению, объем данной книги не позволяет детально рассмотреть все функциональные особенности компонентов и привести достаточное количество примеров их использования. Значительно более развернутое изложение этих вопросов вы можете найти в книгах [2] и [3].

3.2 Компоненты ввода и отображения текстовой информации

3.2.1 Перечень компонентов ввода и отображения текстовой информации

В библиотеке визуальных компонентов C++Builder существует множество компонентов, позволяющих отображать, вводить и редактировать текстовую информацию. В таблице 3.2 приведен их перечень для C++Builder 5 с краткими характеристиками и указанием основных параметров, содержащих отображаемый или вводимый текст. В этой таблице не указаны аналогичные элементы отображения и редактирования текстов, содержащихся в базах данных, так как они будут рассмотрены отдельно в главах 9 и 10.

Таблица 3.2. Компоненты ввода и отображения текстовой информации

Пиктограмма	Компонент	Страница	Описание
	Label (метка)	Standard	Отображение текста, который не изменяется пользователем. Никакого оформления текста не предусмотрено, кроме цвета метки и текста. Основное свойство — Caption .
	StaticText (метка с бордюром)	Additional	Подобен компоненту Label , но обеспечивает возможность задания стиля бордюра. Основное свойство — Caption .
	Panel (панель)	Standard	Компонент является контейнером для группирования органов управления, но может использоваться и для отображения текста с возможностями объемного оформления. Основное свойство — Caption .

Пиктограмма	Компонент	Страница	Описание
	Edit (окно редактирования)	Standard	Отображение, ввод и редактирование однострочных текстов. Имеется возможность оформления объемного бордюра. Основное свойство — Text .
	MaskEdit (окно маскированного редактирования)	Additional	Используется для форматирования данных или для ввода символов в соответствии с шаблоном. Основные свойства — Text и EditText .
	Memo (многострочное окно редактирования)	Standard	Отображение, ввод и редактирование многострочных текстов. Имеется возможность оформления объемного бордюра. Основное свойство — Lines .
	RichEdit (многострочное окно редактирования в формате RTF)	Win32	Компонент представляет собой окно редактирования в стиле Windows в обогащенном формате RTF, позволяющее производить выбор атрибутов шрифта, поиск текста и многое другое. Основное свойство — Lines .
	ListBox (окно списка)	Standard	Отображение стандартного окна списка Windows, позволяющего пользователю выбирать из него пункты. Основное свойство — Items .
	CheckBoxList (список с индикаторами)	Additional	Компонент является комбинацией списка ListBox и индикаторов CheckBox .
	ComboBox (редактируемый список)	Standard	Объединяет функции ListBox и Edit . Пользователь может либо ввести текст, либо выбрать его из списка. Основное свойство — Items .
	StringGrid (таблица строк)	Additional	Отображения текстовой информации в таблице из строк и столбцов с возможностью перемещаться по строкам и столбцам и осуществлять выбор. Основное свойство — Cells .

Помимо перечисленных компонентов отображать текстовые надписи можно непосредственно на свойстве **Canvas** (холст) любого компонента, имеющего это свойство, в частности, непосредственно на форме (см. раздел 5.1.3). Например, оператор вида

```
Canvas.TextOut (60, 16, 'Canvas');
```

обеспечивает печать, начиная с точки с координатами (60, 16), текста « Canvas ». Но это неудобно, так как при этом теряются преимущества визуального проектирования и приходится рассчитывать координаты размещения надписи.

Во всех компонентах шрифт текста, его размер, стиль (жирный, курсив и т.п.), цвет определяются свойством **Font**, которое имеет множество подсвойств, устанавливаемых в процессе проектирования или программно во время выполнения приложения.

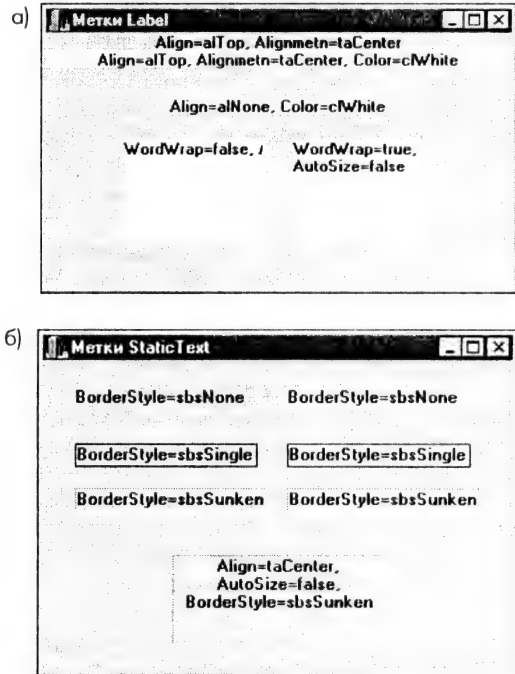
3.2.2 Отображение текста в надписях компонентов Label, StaticText, Panel

Для отображения различных надписей на форме используются в основном компоненты **Label**, **StaticText** и **Panel**. Первые два из этих компонентов — метки, специально предназначенные для отображения текстов. Основное назначение панели **Panel** другое: компоновка компонентов в окне формы. Однако, панель можно использовать и для вывода текстов.

Примеры вывода текста в метки приведены на рис. 3.1. Вид панелей **Panel** вы можете увидеть на рис. 3.38 в разделе 3.7.2.

Рис. 3.1

Примеры вывода текста в компоненты Label (а) и StaticText (б)



Тексты, отображаемые в перечисленных компонентах, определяются значением их свойства **Caption**. Его можно устанавливать в процессе проектирования или задавать и изменять программно во время выполнения приложения. Например:

```
Label1->Caption = "Новый текст";
```

Свойство **Caption** имеет тип строки **AnsiString** (см. соответствующий раздел главы 16). При присваивании этому типу числовой информации происходит ее автоматическое преобразование в строку. Поэтому вы можете непосредственно осуществлять подобные присваивания. Например, оператор

```
Label1->Caption = 5.1;
```

приведет к появлению в метке надписи «5,1». Но если вы хотите занести в метку смешанную информацию, состоящую из строк символов и чисел, вы должны воспользоваться функциями **FloatToStr** и **IntToStr**, переводящими соответственно числа с плавающей запятой и целые в строку. Для формирования текста, состоящего из нескольких фрагментов, можно использовать операцию «+», которая для строк означает их склеивание (конкатенацию). Например, если в программе име-

ется целая переменная **I**, отображающая число сотрудников некоторой организации, то вывести в метку **Label1** информацию об этом можно оператором:

```
Label1->Caption = "Число сотрудников: "+IntToStr(I);
```

Во всех компонентах цвет фона определяется свойством **Color**, а цвет надписи — подсвойством **Color** свойства **Font**. Например, в большинстве меток (кроме верхней) на рис. 3.1 а и в правых метках на рис. 3.1 б задан цвет фона **clWhite** — белый. Если цвет специально не задавать, то цвет фона обычно сливается с цветом контейнера, содержащего метку, так что фон просто не заметен.

Для метки **Label** цвет и шрифт — единственно доступные элементы оформления надписи. Компоненты **StaticText** и **Panel** имеют кроме того свойство **BorderStyle**, определяющее рамку текста — бордюр. На рис. 3.1 б вы можете видеть влияние бордюра на вид метки **StaticText**. При стиле **sbsNone** метка **StaticText** по виду не отличается от метки **Label**. Вероятно, если уж использовать бордюр, то наиболее приятный стиль **sbsSunken**.

Компонент **Panel** кроме свойства **BorderStyle** имеет еще свойства **BevelInner**, **BevelOuter**, **BevelWidth**, **BorderWidth**, которые предоставляют богатые возможности оформления надписи, как вы можете увидеть на рис. 3.38 в разделе 3.7.2. Таким образом, с точки зрения оформления выводимого текста максимальные возможности дает **Panel** и минимальные — **Label**.

Размещение всех рассматриваемых компонентов на форме определяется, в частности, свойствами **Top**, **Left**, **Height**, **Width**, **Aline**, общими для всех оконных компонентов. Эти свойства, определяющие координаты компонента, его размеры и их изменение при изменении пользователем размеров родительского компонента, подробно рассмотрены в разделе 4.2. Так что не будем здесь на этом останавливаться.

Размер меток **Label** и **StaticText** определяется также свойством **AutoSize**. Если это свойство установлено в **true**, то вертикальный и горизонтальный размеры компонента определяются размером надписи. Если же **AutoSize** равно **false**, то выравнивание текста внутри компонента определяется свойством **Alignment**, которое позволяет выравнивать текст по левому краю, правому краю или центру клиентской области метки. В панели **Panel** также имеется свойство **AutoSize**, но оно не относится к размерам надписи **Caption**. Однако, свойство выравнивания **Alignment** работает и для панели.

В метке **Label** имеется свойство **WordWrap** — допустимость переноса слов длинной надписи, превышающей длину компонента, на новую строку. Чтобы такой перенос мог осуществляться, надо установить свойство **WordWrap** в **true**, свойство **AutoSize** в **false** (чтобы размер компонента не определялся размером надписи) и сделать высоту компонента такой, чтобы в нем могло поместиться несколько строк (см. пример правой нижней метки на рис. 3.1 а). Если **WordWrap** не установлено в **true** при **AutoSize** равном **false**, то длинный текст, не помещающийся в рамке метки, просто обрезается (см. пример левой нижней метки на рис. 3.1 а).

В метке **StaticText** перенос длинного текста осуществляется автоматически, если значение **AutoSize** установлено в **false** и размер компонента достаточен для размещения нескольких строк. Для того, чтобы в **StaticText** осуществлялся перенос при изменении пользователем размеров окна, надо осуществлять описанную выше перерисовку компонента методом **Repaint** в обработчике события формы **OnResize**.

В панели размещение надписи в нескольких строках невозможно.

Можно отметить еще одно свойство меток **Label** и **StaticText**, превращающее их в некоторое подобие управляющих элементов. Это свойство **FocusControl** — фокусируемый компонент. Если в свойстве метки **Caption** поместить перед одним из символов символ амперсанта «&», то символ, перед которым поставлен амперсанта, отображается в надписи метки подчеркнутым (сам амперсанта вообще не отображается). Если после этого обратиться к свойству метки **FocusControl**, то из выпадаю-

щего списка можно выбрать элемент, на который будет переключаться фокус, если пользователь нажмет клавиши ускоренного доступа: клавишу **Alt** + подчеркнутый символ. Подобные клавиши ускоренного доступа предусмотрены в управляющих элементах: разделах меню и кнопках. Благодаря свойству **FocusControl** метки могут обеспечить клавишами ускоренного доступа иные элементы, например, окна редактирования (см. раздел 3.2.3), в которых такие клавиши не предусмотрены. Только для того, чтобы клавиши ускоренного доступа в метках срабатывали, необходимо установить свойство **ShowAccelChar** этих меток в **true**.

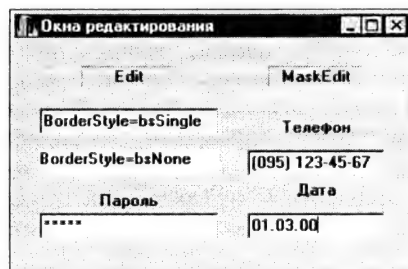
Для отображение текстовой информации, и даже с дополнительной возможностью прокрутки длинных текстов, можно использовать также окна редактирования **Edit** и **MaskEdit** (см. раздел 3.2.3) в режиме **ReadOnly**.

3.2.3 Окна редактирования **Edit** и **MaskEdit**

На рис. 3.2 вы можете увидеть примеры окон редактирования. Внешнее оформление окон редактирования определяется свойством **BorderStyle**, влияние которого на вид компонента вы можете увидеть на том же рисунке.

Рис. 3.2

Примеры окон редактирования



В компонентах **Edit** и **MaskEdit** вводимый и выводимый текст содержится в свойстве **Text** типа **AnsiString**. Это свойство можно устанавливать в процессе проектирования или задавать программно. Выравнивание текста, как это имело место в метках и панелях, невозможно. Перенос строк тоже невозможен. Текст, не помещающийся по длине в окно, просто сдвигается и пользователь может перемещаться по нему с помощью курсора. Свойство **AutoSize** в окнах редактирования имеет смысл, отличный от смысла аналогичного свойства меток: автоматически подстраивается под размер текста только высота, но не ширина окна.

Окна редактирования снабжены многими функциями, свойственными большинству редакторов. Например, в них предусмотрены типичные комбинации «горячих» клавиш: **Ctrl-C** — копирование выделенного текста в буфер обмена (**Clipboard** (команда **Copy**)), **Ctrl-X** — вырезание выделенного текста в буфер **Clipboard** (команда **Cut**), **Ctrl-V** — вставка текста из буфера **Clipboard** в позицию курсора (команда **Paste**), **Ctrl-Z** — отмена последней команды редактирования. Правда, пользователи часто не догадываются об этих возможностях редактирования. Так что полезно напоминать им об этом соответствующими подсказками.

Свойство **AutoSelect** определяет, будет ли автоматически выделяться весь текст при передаче фокуса в окно редактирования. Его имеет смысл задавать равным **true** в случаях, когда при переключении в данное окно пользователь будет вероятнее всего будет заменять текущий текст, а не исправлять его. Имеются также свойства только времени выполнения **SelLength**, **SelStart**, **SelText**, определяющие соответственно длину выделенного текста, позицию перед первым символом выделенного текста и сам выделенный текст. Например, если в окне имеется текст «выделение текста» и в нем пользователь выделил слово «текста», то **SelLength** =

6, **SelStart** = 10 и **SelText** = «текста». Если выделенного текста нет, то свойство **SelStart** просто определяет текущее положение курсора.

Окна редактирования можно использовать и просто как компоненты отображения текста. Для этого надо установить в **true** их свойство **ReadOnly** и целесообразно установить **AutoSelect** в **false**. В этом случае пользователь не сможет изменять отображаемый текст и окно редактирования становится подобным меткам, рассмотренным в разделе 3.2.2. Но имеются и определенные отличия. Во-первых, окна редактирования оформлены несколько иначе (сравните рис. 3.1 и 3.2.). А главное — окна редактирования могут вмещать текст, превышающий их длину. В этом случае пользователь может прокручивать текст, перемещая курсор в окне. Такими особенностями не обладает ни одна метка.

Свойство **Text** окон редактирования имеет тип строки **AnsiString** (см. соответствующий раздел главы 16). При присваивании этому типу числовой информации происходит ее автоматическое преобразование в строку. Поэтому вы можете непосредственно осуществлять подобные присваивания. Например, оператор

```
Edit1->Text = 5. / 2;
```

будет воспринят компилятором нормально и приведет к появлению в окне текста «2,5». Но при вводе из окна числовой информации надо использовать функции **StrToFloat** — преобразование строки в значение с плавающей запятой, и **StrToInt** — преобразование строки в целое значение. Если вводимый текст не соответствует числу (например, содержит недопустимые символы), то функции преобразования генерируют исключение **EConvertError** (см. раздел 12.10 главы 12). Поэтому в программе необходимо предусмотреть обработку этого исключения. Например:

```
int A;
...
try
{
    A = StrToInt(Edit1->Text);
}
catch (EConvertError&)
{
    ShowMessage("Вы ввели ошибочное число; повторите ввод");
}
```

Этот код обеспечивает сообщение пользователю об ошибке ввода и предотвращает ошибочные вычисления. Впрочем, это не лучший вариант предотвратить ошибочный ввод, поскольку пользователь узнает о своей ошибке только после того, как программа пытается использовать введенные данные. В разделе 3.3 описаны специализированные компоненты ввода цифровых данных, которые лучше использовать для ввода чисел. Впрочем, и окно **Edit** можно спроектировать так, что пользователь просто не сможет ввести в него неправильные символы. В главе 4 в разделе 4.3.2 показано, как программно обеспечить, чтобы в окне редактирования пользователь, например, мог вводить только цифры или наоборот — мог вводить символы и не мог вводить числа. В главе 7 в разделе 7.3 рассмотрено, как на основе окна **Edit** можно сделать свой собственный компонент, обладающий этими свойствами.

Свойство **MaxLength** определяет максимальную длину вводимого текста. Если **MaxLength** = 0, то длина текста не ограничена. В противном случае значение **MaxLength** указывает максимальное число символов, которое может ввести пользователь.

Свойство **Modified**, доступное только во время выполнения, показывает, производилось ли редактирование текста в окне. Если вы хотите использовать это свойство, то в момент начала работы пользователя с текстом **Modified** надо установить в **false**. Тогда при последующем обращении к этому свойству можно по его значению (**true** или **false**) установить, было или не было произведено редактирование.

Свойство **PasswordChar** позволяет превращать окно редактирования в окно ввода пароля. По умолчанию значение **PasswordChar** равно "#0" — нулевому символу. В этом случае это обычное окно редактирования. Но если в свойстве указать иной символ (например, символ звездочки "*"), то при вводе пользователем текста в окне будут появляться именно эти символы, а не те, которые вводит пользователь (см. рис. 3.2). Тем самым обеспечивается секретность ввода пароля.

Различные аспекты методики работы с окнами редактирования **Edit** будут еще неоднократно рассматриваться на протяжении этой книги (см. разделы 4.1.8 и 4.3.2.2 главы 4, раздел 7.3 главы 7 и др). Вернемся мы к работе с окном редактирования и в данной главе. В разделе 3.7.8 будет рассказано, как добиться того, что если в окне записан длинный текст, который не виден целиком, то при задержке над этим окном курсора мыши появляется всплывающий ярлычок, содержащий полный текст окна.

Компонент **MaskEdit** отличается от **Edit** тем, что в нем можно задать строку маски в свойстве **EditMask**. Это позволяет обеспечить синтаксически безошибочный ввод пользователем таких данных, как номера телефонов, паспортные данные, адреса, даты, время и т.п. Маска состоит из трех разделов, между которыми ставится точка с запятой «;». В первом разделе — шаблоне записываются специальным образом символы (см. таблицу 3.3), которые можно вводить в каждой позиции, и символы, добавляемые самой маской; во втором разделе записывается 1 или 0 в зависимости от того, надо или нет, чтобы символы, добавляемые маской, включались в свойство **Text** компонента; в третьем разделе указывается символ, используемый для обозначения позиций, в которых еще не осуществлен ввод. Прочитать результат ввода можно или в свойстве **Text**, которое в зависимости от вида второго раздела маски включает или не включает в себя символы маски, или в свойстве **EditText**, содержащем введенный текст вместе с символами маски.

Таблица 3.3. Символы шаблона маски

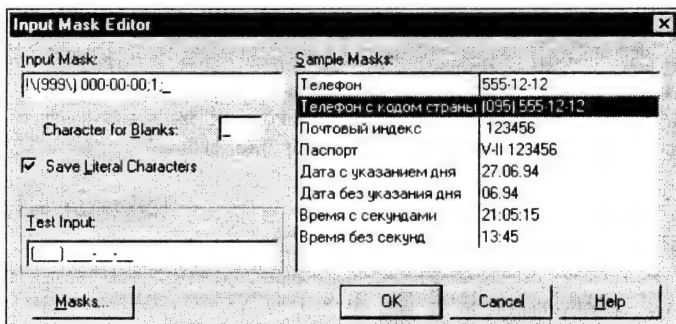
- | | |
|----|---|
| ! | Наличие символа «!» означает, что в EditText недостающие символы <u>предваряются</u> пробелами, а отсутствие символа «!» означает, что пробелы размещаются <u>в конце</u> |
| > | Символ «>» означает, что все последующие за ним символы должны вводиться в верхнем регистре, пока не кончится маска или пока не встретится символ «<» |
| < | Символ «<» означает, что все последующие за ним символы должны вводиться в нижнем регистре, пока не кончится маска или пока не встретится символ «>» |
| <> | Символы «<>» означают, что анализ регистра не производится |
| \ | Символ «\» означает, что следующий за ним символ является буквенным, а не специальным, характерным для маски. Например, символ «>» после символа «\» воспримется как знак > (больше), а не как символ, указывающий на верхний регистр |
| L | Символ «L» означает, что в данной позиции должна быть буква |
| l | Символ «l» означает, что в данной позиции может быть только буква или ничего |
| A | Символ «A» означает, что в данной позиции должна быть буква или цифра |
| a | Символ «a» означает, что в данной позиции может быть буква, или цифра, или ничего |

- С** Символ «С» означает, что в данной позиции должен быть любой символ
- с** Символ «с» означает, что в данной позиции может быть любой символ или ничего
- 0** Символ «0» означает, что в данной позиции должна быть цифра
- 9** Символ «9» означает, что в данной позиции может быть цифра или ничего
- #** Символ «#» означает, что в данной позиции может быть цифра, знак «+», знак «-» или ничего
- :** Символ «:» используется для разделения часов, минут и секунд
- /** Символ «/» используется для разделения месяцев, дней и годов в датах
- _** Символ «_» означает автоматическую вставку в текст пробела

Вводить маску можно непосредственно в свойство **EditMask**. Но удобнее пользоваться специальным редактором масок, вызываемым при нажатии кнопки с многоточием в строке свойства **EditMask** в Инспекторе Объектов. Окно редактора масок имеет вид, представленный на рис. 3.3 (на рисунке изображены маски файла российского шаблона, описанного далее).

Рис. 3.3

Окно редактора масок с загруженным файлом российских стандартных масок



В редакторе масок окно **Sample Masks** содержит наименования стандартных масок и примеры ввода с их помощью. В окно **Input Mask** надо ввести маску. Если вы выбираете одну из стандартных масок, то окно **Input Mask** автоматически заполняется и вы можете, если хотите, отредактировать эту маску.

Окно **Character for Blanks** определяет символ, используемый для обозначения позиций, в которых еще не осуществлен ввод (третий раздел маски). Индикатор **Save Literal Characters** определяет второй раздел маски: установлен, если второй раздел равен 1, и не установлен, если второй раздел равен 0.

Кнопка **Masks** позволяет выбрать и загрузить какой-либо другой файл стандартных масок. К сожалению, среди файлов стандартных масок, поставляемых с C++Builder, отсутствует маска, соответствующая российским стандартам. Но вы легко можете сами сделать себе такой файл стандартных масок. Он делается в обычном текстовом редакторе и должен сохраняться как «только текст» с расширением **.dem**. Чтобы редактор масок C++Builder видел этот файл, его надо сохранить в каталоге C++Builder BIN. Каждая строка файла состоит из трех частей, разделяемых символом вертикальной черты. Первая часть состоит из пояснительного текста, появляющегося в левой панели окна **Sample Masks** редактора масок. Вторая часть — пример, который появляется в правой панели окна **Sample Masks** редактора

масок. А третья часть — сама маска. Например, я сделал себе файл с текстом, приведенным ниже, и сохранил его с именем **ru.dem**.

```
Телефон | 5551212 | !000-00-00;0;_  
Телефон с кодом страны | 0955551212 | !\ (999\ ) 000-00-00;0;_  
Почтовый индекс | 123456 | !0000000;1;_  
Паспорт | VII123456 | !L-LL 999999;0;_  
Дата с указанием дня | 270694 | !99/99/00;1;_  
Дата без указания дня | 0694 | !99/00;1;_  
Время с секундами | 210515 | !90:00:00;1;_  
Время без секунд | 1345 | !90:00;1;_
```

На рис. 3.3 вы можете видеть его в работе, а на рис. 3.2 вы можете видеть ввод в окна с масками телефона и даты.

Рассмотрим примеры масок. В приведенном выше файле маска для ввода номера телефона имеет вид:

```
!\ (999\ ) 000-00-00;0;_
```

В этой маске символ «9» означает, что в соответствующей позиции может быть только цифра. Символ «0» означает, что в данной позиции должна быть цифра. Символ подчеркивания в конце маски будет заполнять пустые позиции. Таким образом, пользователю для ввода в окне будет отображен шаблон (см. рис. 3.2):

```
( ) _ _ - _ _ _
```

Поскольку второй раздел маски равен 0, то при чтении введенных пользователем значений свойства **EditText** и **Text** будут различаться. Свойство **EditText** для примера рис. 3.2 будет равно «(095) 123-45-67», а свойство **Text** будет равно «0951234567». Если второй раздел маски сделать равным 1, то значения обоих свойств будут равны «(095) 123-45-67».

Рассмотрим еще пример. Если с помощью **EditMask** надо ввести, например, целое число без знака, состоящее не более, чем из двух цифр, можно задать маску «99;0;». Если число обязательно должно быть двузначным, то маска должна иметь вид «00;0;».

3.2.4 Многострочные окна редактирования Memo и RichEdit

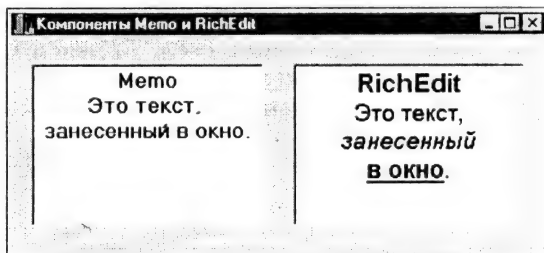
Компоненты **Memo** и **RichEdit** (см. пример на рис. 3.4) являются окнами редактирования многострочного текста. Они так же, как и окно **Edit**, снабжены многими функциями, свойственными большинству редакторов. В них предусмотрены типичные комбинации «горячих» клавиш: Ctrl-C — копирование выделенного текста в буфер обмена Clipboard (команда Copy), Ctrl-X — вырезание выделенного текста в буфер Clipboard (команда Cut), Ctrl-V — вставка текста из буфера Clipboard в позицию курсора (команда Paste), Ctrl-Z — отмена последней команды редактирования.

В компоненте **Memo** формат (шрифт, его атрибуты, выравнивание) одинаков для всего текста и определяется свойством **Font**. Если вы сохраните в файле текст, введенный или отредактированный пользователем, то будет создан текстовый файл, содержащий только символы и не содержащий элементов форматирования. При последующем чтении этого файла в **Memo** формат будет определяться текущим состоянием свойства **Font** компонента **Memo**, а не тем, в каком формате ранее вводился текст.

Компонент **RichEdit** работает с текстом в обогащенном формате RTF. При желании изменить атрибуты вновь вводимого фрагмента текста вы можете задать свойство **SelAttributes**. Это свойство типа **TTextAttributes**, которое в свою очередь имеет подсвойства: **Color** (цвет), **Name** (имя шрифта), **Size** (размер), **Style** (стиль) и ряд других. Например, введите на форму компонент **RichEdit**, диалог выбора шрифта **FontDialog** со страницы Dialogs (см. подробнее в разделе 3.8.4) и кнопку

Рис. 3.4

Примеры компонентов Memo и RichEdit



Button, которая позволит пользователю менять атрибуты текста. В обработчик щелчка кнопки введите текст:

```
if(FontDialog1->Execute())
    RichEdit1->SelAttributes->Assign(FontDialog1->Font);
RichEdit1->SetFocus();
```

Запустите приложение и увидите, что вы можете менять атрибуты текста, выполняя отдельные фрагменты различными шрифтами, размерами, цветами, стилями. Устанавливаемые атрибуты влияют на выделенный текст или, если ничего не выделено, то на атрибуты нового текста, вводимого начиная с текущей позиции курсора (позиция курсора определяется свойством **SelStart**).

В компоненте имеется также свойство **DefAttributes**, содержащее атрибуты по умолчанию. Эти атрибуты действуют до того момента, когда изменяются атрибуты в свойстве **SelAttributes**. Но значения атрибутов в **DefAttributes** сохраняются и в любой момент эти значения могут быть методом **Assign** присвоены атрибутам свойства **SelAttributes**, чтобы вернуться к прежнему стилю.

Свойство **DefAttributes** доступно только во время выполнения. Поэтому его атрибуты при необходимости можно задавать, например, в обработчике события **OnCreate**.

За выравнивание, отступы и т.д. в пределах текущего абзаца отвечает свойство **Paragraph** типа **TParaAttributes**. Этот тип в свою очередь имеет ряд свойств:

Alignment	Определяет выравнивание текста. Может принимать значения taLeftJustify (влево), taCenter (по центру) или taRightJustify (вправо)
FirstIndent	Число пикселей отступа красной строки
Numbering	Управляет вставкой маркеров, как в списках. Может принимать значения nsNone — отсутствие маркеров, nsBullet — маркеры ставятся
LeftIndent	Отступ в пикселях от левого поля
RightIndent	Отступ в пикселях от правого поля
TabCount	Количество позиций табуляции
Tab	Значения позиций табуляции в пикселях

Значения подсвойств свойства **Paragraph** можно задавать только в процессе выполнения приложения, например, в событии создания формы или при нажатии какой-нибудь кнопки. Значения подсвойств свойства **Paragraph** относятся к тому абзацу, в котором находится курсор. Например, каждый из следующих операторов осуществит соответственное выравнивание текущего абзаца:


```
RichEdit1->Paragraph->Alignment = taLeftJustify; // Влево
RichEdit1->Paragraph->Alignment = taCenter;      // По центру
RichEdit1->Paragraph->Alignment = taRightJustify; // Вправо
```

Следующий оператор приведет к тому, что текущий абзац будет отображаться как список, т.е. с маркерами:

```
RichEdit1->Paragraph->Numbering = nsBullet;
```

Уничтожение списка в текущем абзаце осуществляется оператором

```
RichEdit1->Paragraph->Numbering = nsNone;
```

В целом, если с помощью компонента **ActionList** (см. раздел 3.9.1) определено некоторое действие ввода и уничтожения списка, названное **ABullet**, то операторы обработки соответствующего действия могут иметь вид:

```
if (ABullet->Checked)
    RichEdit1->Paragraph->Numbering = nsBullet;
else RichEdit1->Paragraph->Numbering = nsNone;
ABullet->Checked = ! ABullet->Checked;
```

Они обеспечивают переключение соответствующей быстрой кнопки и раздела меню из нажатого состояния (отмеченного) в ненажатое с соответствующим изменением вида текущего абзаца.

На рис. 3.5 приведен пример текстового редактора, использующего описанные выше свойства компонента **RichEdit**. Текст в окне редактора частично поясняет атрибуты шрифта, использованные при его написании.

Свойства **TabCount** и **Tab** имеют смысл при вводе текста только при значении свойства компонента **WantTabs** = **true**. Это свойство разрешает пользователю вводить в текст символ табуляции. Если **WantTabs** = **false**, то нажатие пользователем клавиши табуляции просто переключит фокус на очередной компонент и символ табуляции в текст не введется.

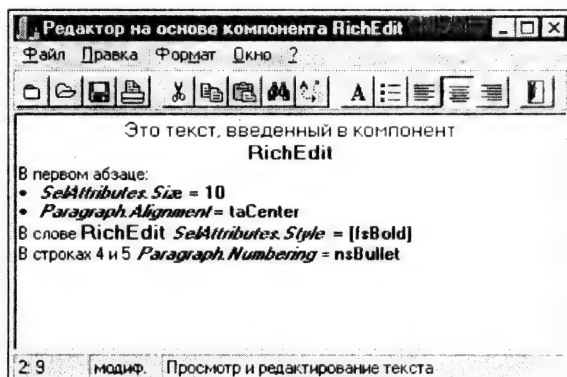
Мы рассмотрели основные отличия **Memo** и **RichEdit**. Теперь остановимся на общих свойствах этих окон редактирования.

Свойства **Alignment** и **WordWrap** имеют тот же смысл, что, например, в метках, и определяют выравнивание текста и допустимость переноса длинных строк. Установка свойства **ReadOnly** в **true** задает текст только для чтения. Свойство **MaxLength** определяет максимальную длину вводимого текста. Если **MaxLength** = 0, то длина текста не ограничена. Свойства **WantReturns** и **WantTab** определяют допустимость ввода пользователем в текст символов перевода строки и табуляции.

Свойство **ScrollBars** определяет наличие полос прокрутки текста в окне. По умолчанию **ScrollBars** = **ssNone**, что означает их отсутствие. Пользователь может в этом случае перемещаться по тексту только с помощью курсора. Можно задать

Рис. 3.5

Пример редактора на основе компонента **RichEdit**

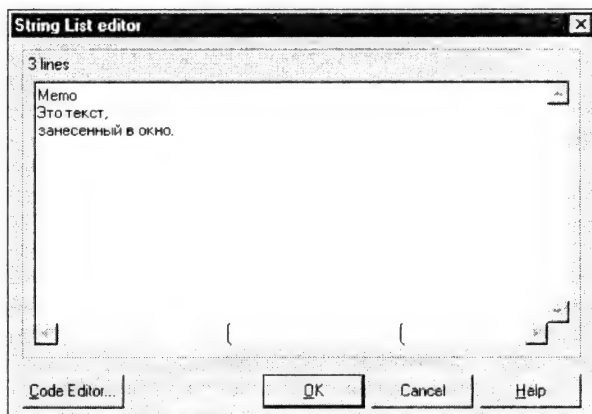


свойству **ScrollBars** значения **ssHorizontal**, **ssVertical** или **ssBoth**, что будет соответственно означать наличие горизонтальной, вертикальной или обеих полос прокрутки.

Основное свойство окон **Memo** и **RichEdit** — **Lines**, содержащее текст окна в виде списка строк и имеющее тип **TStrings**. Начальное значение текста можно установить в процессе проектирования, нажав кнопку с многоточием около свойства **Lines** в окне Инспектора Объектов. Перед вами откроется окно редактирования списков строк, представленное на рис. 3.6. Вы можете редактировать или вводить текст непосредственно в этом окне, или нажать кнопку **CodeEditor** и работать в обычном окне Редактора Кода. В этом случае, завершив работу с текстом, выберите из контекстного меню, всплывающего при щелчке правой кнопкой мыши, команду **Close Page** и ответьте утвердительно на вопрос, хотите ли вы сохранить текст в соответствующем свойстве окна редактирования.

Рис. 3.6

Окно редактирования списков строк



Во время выполнения приложения вы можете заносить текст в окно редактирования с помощью методов свойства **Lines** типа **TStrings**. Этот тип широко используется в свойствах многих компонентов и его подробное описание вы можете найти в разделе 16.4 главы 16. Здесь коротко укажем только на его основные свойства и методы, используемые в свойстве **Lines**.

Весь текст, представленный одной строкой типа **String**, внутри которой используются разделители типа символов возврата каретки и перевода строки, содержится в свойстве **Text**.

Доступ к отдельной строке текста вы можете получить с помощью свойства **AnsiString Strings[int Index]**. Индексы, как и везде в C++Builder, начинаются с 0. Так что **Memo1->Lines->Strings[0]** — это текст первой строки. Учтите, что если окно редактирования изменяется в размерах при работе с приложением и свойство **WordWrap = true**, то индексы строк будут изменяться при переносах строк, так что в этих случаях индекс мало о чем говорит.

Свойство только для чтения **Count** указывает число строк в тексте.

Для очистки текста в окне надо выполнить процедуру **Clear**. Этот метод относится к самому окну, а не к его свойству **Lines**.

Для занесения новой строки в конец текста окна редактирования можно воспользоваться методами **Add** или **Append** свойства **Lines**. Для загрузки текста из файла применяется метод **LoadFromFile**. Сохранение текста в файле осуществляется методом **SaveToFile**.

Пусть, например, в вашем приложении имеется окно редактирования **Edit1**, в котором пользователь вводит имя сотрудника, и есть кнопка, при щелчке на кото-

рой в окно **Memo1** должна занестись шапка характеристики этого сотрудника, после чего пользователь может заполнить текст характеристики.

Обработчик щелчка на кнопке может иметь вид:

```
Memol->Clear();
Memol->Lines->Add("Х А Р А К Т Е Р И С Т И К А");
Memol->Lines->Add("Сотрудник "+Edit1->Text);
Memol->SetFocus();
```

В компоненте **RichEdit** тот же фрагмент может выглядеть иначе. Можно, например, строки «Характеристика» и «Сотрудник ...» выделить жирным шрифтом и выровнять по центру, после чего вернуться к стилю по умолчанию. Код, выполняющий подобные операции, может иметь вид:

```
RichEdit1->Clear();
/* установка выравнивания по центру */
RichEdit1->Paragraph->Alignment = taCenter;
/* установка жирного шрифта */
RichEdit1->SelAttributes->Style =
RichEdit1->SelAttributes->Style << fsBold;
RichEdit1->Lines->Add("Х А Р А К Т Е Р И С Т И К А");
RichEdit1->Lines->Add("Сотрудник "+Edit1->Text);
/* восстановление атрибутов по умолчанию */
RichEdit1->SelAttributes->Assign(RichEdit1->DefAttributes);
/* установка выравнивания по левому краю */
RichEdit1->Paragraph->Alignment = taLeftJustify;
RichEdit1->SetFocus();
```

Загрузка в окно **RichEdit1** текста из файла (например, хранящейся в файле характеристики сотрудника) может осуществляться командой

```
RichEdit1->Lines->LoadFromFile("text.rtf");
```

Сохранение текста в файле может осуществляться командой

```
RichEdit1->Lines->SaveToFile("text.rtf");
```

Свойство **SelStart** компонентов **Memo** и **RichEdit** указывает позицию курсора в тексте или начало выделенного пользователем текста. Свойство **CaretPos** указывает на структуру, поле **X** которой содержит индекс символа в строке, перед которым расположен курсор, а поле **Y** — индекс строки, в которой находится курсор. Таким образом, учитывая, что индексы начинаются с 0, значения **RichEdit1->CaretPos.y + 1** и **RichEdit1->CaretPos.x + 1** определяют соответственно номер строки и символа в ней, перед которым расположен курсор. В редакторе на рис. 3.5 именно эти значения использованы, чтобы отображать в полосе состояния (см. раздел 3.7.7) позицию курсора.

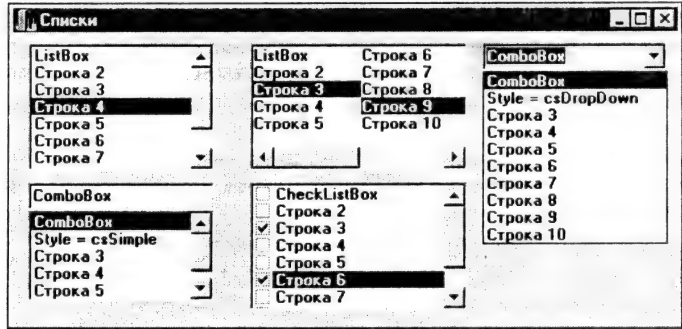
3.2.5 Компоненты выбора из списков — ListBox, CheckListBox, ComboBox

Пример компонентов **ListBox**, **CheckListBox**, **ComboBox**, обеспечивающих выбор из списка, приведен на рис. 3.7.

Компоненты **ListBox** и **ComboBox** отображают списки строк. Использование этих компонентов позволяет обеспечить безошибочный ввод информации пользователем в тех случаях, когда он должен выбрать ответ из конечного множества альтернатив, например, из списка отделов предприятия. Компоненты списков отличаются друг от друга прежде всего тем, что **ListBox** только отображает данные и позволяет пользователю выбрать из них то, что ему надо, а **ComboBox** позволяет также редактировать данные. Кроме того различается форма отображения списков. **ListBox** отображает список в раскрытом виде и автоматически добавляет в

Рис. 3.7

Пример компонентов выбора из списков



список полосы прокрутки, если все строки не помещаются в окне компонента. **ComboBox** позволяет отображать список как в развернутом виде, так и в виде выпадающего списка, что обычно удобнее, так как экономит площадь окна приложения.

Основное свойство обоих компонентов, содержащее список строк, — **Items**, имеющее рассмотренный ранее тип **TStrings**. Заполнить его во время проектирования можно, нажав кнопку с многоточием около этого свойства в окне Инспектора Объектов. При этом открывается окно редактирования, рассмотренное ранее в разделе 3.2.4 (рис. 3.6). Каждая записанная в нем строка будет соответствовать строке списка. Во время выполнения работать со свойством **Items** можно, пользуясь свойствами и методами класса **TStrings** (см. раздел 3.2.4 и главу 16) — **Clear**, **Add** и другими. Свойство **Items->Count** определяет текущее число строк списка.

В компоненте **ListBox** имеется свойство **MultiSelect**, разрешающее пользователю множественный выбор в списке (на рис. 3.7 это свойство установлено в **true** в среднем верхнем списке). Если **MultiSelect = false** (значение по умолчанию), то пользователь может выбрать только один элемент списка. В этом случае можно узнать индекс выбранной строки из свойства **ItemIndex**, доступного только во время выполнения. Если ни одна строка не выбрана, то **ItemIndex = -1**. Начальное значение **ItemIndex** невозможно задать во время проектирования. По умолчанию **ItemIndex = -1**. Это означает, что ни один элемент списка не выбран. Если вы хотите задать этому свойству какое-то другое значение, т.е. установить выбор по умолчанию, который будет показан в момент начала работы приложения, то сделать это можно, например, в обработчике события **OnCreate** формы, введя в него оператор вида

```
ListBox1->ItemIndex = 0;
```

Если допускается множественный выбор (**MultiSelect = true**), то значение **ItemIndex** соответствует тому элементу списка, который находится в фокусе. При множественном выборе проверить, выбран ли данный элемент, можно проверив свойство **bool Selected[int Index]**.

На способ множественного выбора при **MultiSelect = true** влияет свойство **ExtendedSelect**. Если **ExtendedSelect = true**, то пользователь может выделить интервал элементов, выделив один из них, затем нажав клавишу **Shift** и переведя курсор к другому элементу. Выделить не прилегающие друг к другу элементы пользователь может, если будет удерживать во время выбора нажатой клавишу **Ctrl**. Если же **ExtendedSelect = false**, то клавиши **Shift** и **Ctrl** при выборе не работают.

Свойство **Columns** определяет число столбцов, в которых будет отображаться список, если он не помещается целиком в окне компонента **ListBox** (в среднем верхнем списке на рис. 3.7 свойство **Columns** равно 2).

Свойство **Sorted** позволяет упорядочить список по алфавиту. При **Sorted = true** новые строки в список добавляются не в конец, а по алфавиту.

Свойство **Style**, установленное в **lbStandard** (значение по умолчанию) соответствует списку строк. Другие значения **Style** позволяют отображать в списке не только текст, но и изображения (эти стили мы рассматривать не будем — см. в книге [2]).

Имеется еще один компонент, очень похожий на **ListBox** — это список с индикаторами **CheckListBox**. Выглядит он так же, как **ListBox** (средний нижний список на рис. 3.7), но около каждой строки имеется индикатор, который пользователь может переключать. Индикаторы можно переключать и программно, если список используется для вывода данных и необходимо в нем отметить какую-то характеристику каждого объекта, например, наличие товара данного наименования на складе.

Все свойства, характеризующие компонент **CheckListBox** как список, аналогичны **ListBox**, за исключением свойств, определяющих множественный выбор. Эти свойства компоненту **CheckListBox** не нужны, поскольку в нем множественный выбор можно осуществлять установкой индикаторов. А свойства, связанные с индикаторами, описаны в разделе 3.5.4.

Рассмотрим теперь компонент **ComboBox**. Стиль изображения этого компонента определяется свойством **Style**, которое может принимать следующие основные значения:

csDropDown	Выпадающий список со строками одинаковой высоты и с окном редактирования, позволяющим пользователю вводить или редактировать текст (правый список на рис. 3.7)
csSimple	Развернутый список со строками одинаковой высоты и с окном редактирования, позволяющим пользователю вводить или редактировать текст (левый нижний список на рис. 3.7)
csDropDownList	Выпадающий список со строками одинаковой высоты, не содержащий окна редактирования
csOwnerDrawFixed	Выпадающий список типа csDropDown с графической прорисовкой элементов одинаковой высоты
csOwnerDrawVariable	Выпадающий список типа csDropDown с графической прорисовкой элементов, которые могут иметь различную высоту

Выбор пользователя или введенный им текст можно определить по значению свойства **Text**. Если же надо определить индекс выбранного пользователем элемента списка, то можно воспользоваться обсуждавшимся в компоненте **ListBox** свойством **ItemIndex**, доступным только во время выполнения. Все сказанное ранее об **ItemIndex** и о задании его значения по умолчанию справедливо и для компонента **ComboBox**. Причем для **ComboBox** задание начального значения **ItemIndex** еще актуальнее, чем для **ListBox**. Если начальное значение не задано, то в момент запуска приложения окно редактирования списка будет отображать не один из элементов списка, а значение свойства **Text**. Значит надо или вводить в приложение оператор, задающий в первый момент значение **ItemIndex**, или вводить во время проектирования в свойство **Text** какое-то приглашение к дальнейшим действиям. Иначе пользователь будет в недоумении, что надо делать с этим списком.

Если в окне проводилось редактирование данных, то **ItemIndex** = -1. По этому признаку можно определить, что редактирование проводилось.

Свойство **MaxLength** определяет максимальное число символов, которые пользователь может ввести в окно редактирования. Если **MaxLength** = 0, то число вводимых символов не ограничено.

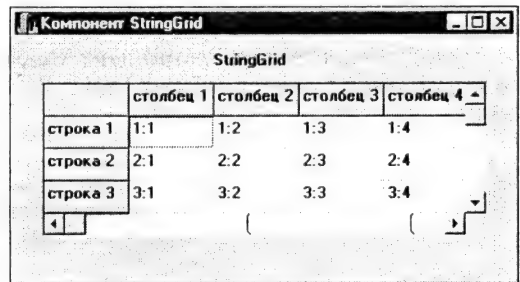
Как и в компоненте **ListBox**, свойство **Sorted** позволяет упорядочить список по алфавиту. При **Sorted** = **true** новые строки в список добавляются не в конец, а по алфавиту.

3.2.6 Таблица строк — компонент StringGrid

Компонент **StringGrid** (см. пример на рис. 3.8) представляет собой таблицу, содержащую строки. Данные таблицы могут быть только для чтения или редактируемы. Таблица может иметь полосы прокрутки, причем заданное число первых строк и столбцов может быть фиксированным и не прокручиваться. Таким образом, можно задать заголовки столбцов и строк, постоянно присутствующие в окне компонента. Каждой ячейке таблицы может быть поставлен в соответствие некоторый объект.

Рис. 3.8

Пример компонента StringGrid



Компонент **StringGrid** предназначен в первую очередь для отображения таблиц текстовой информации. Однако в разделе 3.4.4 поясняется, как этот компонент может отображать и графическую информацию.

Основные свойства компонента, определяющие отображаемый текст:

System::AnsiString Cells [int ACol][int ARow]	Строка, содержащаяся в ячейке с индексами столбца и строки ACol и ARow
Classes::TStrings* Cols [int Index]	Список строк и связанных с ними объектов, содержащихся в столбце с индексом Index
Classes::TStrings* Rows [int Index]	Список строк и связанных с ними объектов, содержащихся в строке с индексом Index
System::TObject* Objects [int ACol][int ARow]	Объект, связанный со строкой, содержащейся в ячейке с индексами столбца и строки ACol и ARow

Все эти свойства доступны во время выполнения. Задавать тексты можно программно или по отдельным ячейкам, или сразу по столбцам и строкам с помощью методов класса **TStrings** (см. краткое описание в разделе 3.2.4 и подробное — в главе 16).

Свойства **ColCount** и **RowCount** определяют соответственно число столбцов и строк, свойства **FixedCols** и **FixedRows** — число фиксированных, не прокручиваемых столбцов и строк. Цвет фона фиксированных ячеек определяется свойством **FixedColor**. Свойства **LeftCol** и **TopRow** определяют соответственно индексы первого видимого на экране в данный момент прокручиваемого столбца и первой видимой прокручиваемой строки.

Свойство **ScrollBars** определяет наличие в таблице полос прокрутки. Причем полосы прокрутки появляются и исчезают автоматически в зависимости от того, помещается таблица в соответствующий размер, или нет.

Свойство **Options** является множеством, определяющим многие свойства таблицы: наличие разделительных вертикальных и горизонтальных линий в фиксированных (**goFixedVertLine** и **goFixedHorzLine**) и не фиксированных (**goVertLine** и **goHorzLine**) ячейках, возможность для пользователя изменять с помощью мыши размеры столбцов и строк (**goColSizing** и **goRowSizing**), перемещать столбцы и строки (**goColMoving** и **goRowMoving**) и многое другое. Важным элементом в свойстве **Options** является **goEditing** — возможность редактировать содержимое таблицы.

В основном компонент **StringGrid** используется для выбора пользователем каких-то значений, отображенных в ячейках. Свойства **Col** и **Row** показывают индексы столбца и строки выделенной ячейки. Возможно также выделение пользователем множества ячеек, строк и столбцов.

Среди множества событий компонента **StringGrid** следует отметить событие **OnSelectCell**, возникающее в момент выбора пользователем ячейки. В обработчик этого события передаются целые параметры **ACol** и **ARow** — столбец и строка выделенной ячейки, и булев параметр **CanSelect** — допустимость выбора. Параметр **CanSelect** можно использовать для запрета выделения ячейки, задав его значение **false**. А параметры **ACol** и **ARow** могут использоваться для какой-то реакции программы на выделение пользователя. Например, оператор

```
Label1->Caption = "Выбрана ячейка " + IntToStr(ARow) +  
                ':' + IntToStr(ACol);
```

выдаст в метку **Label1** сообщение о строке и столбце выбранной ячейки. А оператор

```
Label1->Caption = StringGrid1->Cells[ACol][ARow];
```







выведет в ту же метку текст выделенной ячейки. Конечно, в реальном приложении задача заключается не в том, чтобы вывести подобные тексты при выборе пользователем той или иной ячейки, а в том, чтобы сделать нечто более полезное.

3.3 Ввод и отображение чисел, дат и времени

3.3.1 Перечень компонентов ввода и отображения чисел, дат и времени

В библиотеке визуальных компонентов C++Builder существует ряд компонентов, позволяющих вводить, отображать и редактировать числа, даты и время. Конечно, с подобной информацией можно обращаться и просто как с текстовой, используя компоненты, описанные в разделе 3.2. Но это не удобно, так как не гарантирует от ошибок при вводе. В таблице 3.4 приведен перечень специализированных компонентов ввода и отображения чисел, дат и времени с краткими характеристиками и указанием основных параметров, содержащих отображаемый или вводимый текст.

Таблица 3.4. Компоненты ввода и отображения чисел, дат и времени

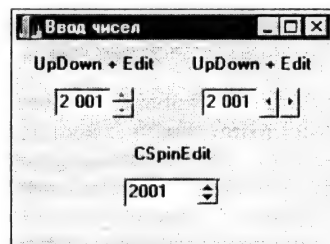
Пиктограмма	Компонент	Страница	Описание
	UpDown (кнопка-счетчик)	Win32	Кнопка-счетчик, в сочетании с компонентами Edit и другими позволяющая вводить цифровую информацию. Основное свойство — Position
	SpinEdit (кнопка-счетчик с окном редактирования)	Samples	Окно редактирования в комбинации с кнопкой-счетчиком. Почти то же, что комбинация Edit и UpDown . Основное свойство — Value .
	DateTimePicker (окно ввода дат и времени)	Win32	Ввод даты (с выпадающим календарем) и времени. Основные свойства — Date и Time
	MonthCalendar (окно ввода дат)	Win32	Ввод дат с выбором из календаря
	Calendar (календарь на указанный месяц)	Samples	Отображение календаря на указанный месяц. Компонент DateTimePicker имеет больше возможностей по вводу дат, чем этот компонент. Основные свойства — Month и Day
	F1Book (страницы Excel)	ActiveX	Компонент ввода и обработки числовой информации, аналогичный страницам Excel

3.3.2 Ввод и отображение целых чисел — компоненты UpDown и SpinEdit

В C++Builder имеются специализированные компоненты, обеспечивающие ввод целых чисел — **UpDown** и **SpinEdit** (см. пример на рис. 3.9).

Рис. 3.9

Пример компонентов UpDown и SpinEdit



Компонент **UpDown** превращает окно редактирования **Edit** в компонент, в котором пользователь может выбирать целое число, изменяя его кнопками со стрелками. Если к тому же установить в **true** свойство окна **ReadOnly**, то пользователь просто не сможет ввести в окно какой-либо свой текст и вынужден будет ограничиться выбором числа. Компонент **SpinEdit** представляет собой сочетание **Edit** и **UpDown**, оформленное как отдельный тип компонента.

Основное свойство компонента **UpDown** — **Associate**, связывающее кнопки со стрелками с одним из оконных компонентов, обычно с **Edit**. Чтобы опробовать компонент **UpDown**, перенесите на форму его и окно редактирования **Edit**, распо-

ложив **Edit** там, где это требуется, а **UpDown** — в любом месте формы. Далее в выпадающем списке свойства **Associate** компонента **UpDown** выберите **Edit1**. Компонент **UpDown** немедленно переместится к **Edit** и как бы сольется с ним.

Свойство **AlignButton** компонента **UpDown**, которое может принимать значения **udLeft** или **udRight**, определяет, слева или справа от окна будут размещаться кнопки. Свойство **Orientation**, которое может принимать значения **udHorizontal** или **udVertical**, определяет, расположатся ли кнопки по вертикали (одна под другой — см. левый компонент на рис. 3.9) или по горизонтали (одна рядом с другой — см. правый компонент на рис. 3.9). Свойство **ArrowKeys** определяет, будут ли управлять компонентом клавиши клавиатуры со стрелками. Свойство **Thousands** определяет наличие или отсутствие разделительного пробела между каждыми тремя цифрами разрядов вводимого числа.

Свойства **Min** и **Max** компонента **UpDown** задают соответственно минимальное и максимальное значения чисел, свойство **Increment** задает приращение числа при каждом нажатии на кнопку. Свойство **Position** определяет текущее значение числа. Это свойство можно читать, чтобы узнать, какое число ввел пользователь. Его можно задать во время проектирования в диапазоне **Min** — **Max**. Тогда это будет значение числа по умолчанию, отображаемое в окне в начале выполнения приложения.

Свойство **Wrap** определяет, как ведет себя компонент при достижении максимального или минимального значений. Если **Wrap = false**, то при увеличении или уменьшении числа до максимального или минимального значения это число фиксируется на предельном значении и нажатие кнопки, пытающейся увеличить максимальное число или уменьшить минимальное, ни к чему не приводит. Если же **Wrap = true**, то попытка превысить максимальное число приводит к его сбросу на минимальное значение. Аналогично, попытка уменьшить минимальное число приводит к его сбросу на максимальное значение. Т.е. изменение чисел «закольцовывается».

Если в компоненте **Edit**, связанном с **UpDown**, не задать **ReadOnly** равным **true**, то пользователь сможет редактировать число, не пользуясь кнопками со стрелками. Это удобно, если требуемое число далеко от указанного по умолчанию, а шаг приращения **Increment** в **UpDown** мал. Но тут проявляется серьезный недостаток компонента **UpDown**: ничто не мешает пользователю ввести по ошибке не цифры, а какие-то другие символы. Чтобы избавиться от этого недостатка, можно использовать прием, описанный в главе 4 в разделе 4.3.2.2, который не дает возможность пользователю ввести в окно редактирования какие-то символы, кроме цифр. Но лучше для этих целей использовать компонент **SpinEdit**.

Свойства компонента **SpinEdit** похожи на рассмотренные, только имеют другие имена: свойства **Min**, **Max**, **Position** называются соответственно **MinValue**, **MaxValue**, **Value**. В целом компонент **SpinEdit** во многих отношениях удобнее простого сочетания **UpDown** и **Edit**. Так что, если не требуются какие-то из описанных выше дополнительных возможностей **UpDown** (нестандартное расположение кнопок, «закольцовывание» изменений и т.п.), то можно рекомендовать пользоваться компонентом **SpinEdit**.

3.3.3 Ввод и отображение дат и времени — компоненты **DateTimePicker**, **MonthCalendar**, **Calendar**

Примеры компонентов ввода и отображение дат и времени приведены на рис. 3.10.

Из этих компонентов наиболее удобным является **DateTimePicker** (на рис. 3.10, слева сверху показан этот компонент в режиме ввода времени, а ниже — в двух вариантах режима ввода даты). Компонент очень эффектен за счет появления выпадающего календаря (иногда даже слишком эффектен для строго оформ-

Рис. 3.10

Примеры компонентов отображения дат и времени



ленного приложения) и обеспечивает безошибочный с точки зрения синтаксиса ввод дат и времени. Его свойство **Kind** определяет режим работы компонента: **dtkDate** — ввод даты, **dtkTime** — ввод времени.

При вводе дат можно задать свойство **DateMode** равным **dmComboBox** — наличие выпадающего календаря, или равным **dmUpDown** — наличие кнопок увеличения и уменьшения (см. средний компонент **DateTimePicker** на рис. 3.10), напоминающих те, которые используются в описанных ранее компонентах **UpDown** и **SpinEdit**. Только в данном случае пользователь может независимо устанавливать с помощью кнопок число, месяц и год. Формат представления дат определяется свойством **DateFormat**, которое может принимать значения **dfShort** — краткий формат (например, «08.03.00»), или **dfLong** — полный формат (например, «8 Март 2000г.»).

Значение даты по умолчанию можно задать в Инспекторе Объектов через свойство **Date**. Это же свойство читается для определения заданной пользователем даты. При чтении **Date** надо учитывать тип этого свойства — **TDateTime**, представляющий собой число с плавающей запятой, целая часть которого содержит число дней, отсчитанное от некоторого начала календаря, а дробная часть равна части 24-часового дня, т.е. характеризует время и не относится к дате. За начало календаря принята дата 12/30/1899 00 часов.

Для преобразования значения свойства **Date** в строку можно воспользоваться функцией **DateToStr**. Например, оператор

```
Memol->Lines->Add("Дата: " + DateToStr(DateTimePicker1->Date));
```

добавит в окно **Memol** строку вида «Дата: 08.03.00».

При вводе дат можно задать значения свойств **MaxDate** и **MinDate**, определяющих соответственно максимальную и минимальную дату, которую может задать пользователь.

В режиме ввода времени **dtkTime** введенное пользователем значение можно найти в свойстве **Time**, тип которого — тот же рассмотренный выше **TDateTime**. Преобразовать время в строку можно функцией **TimeToStr**.

Компонент **MonthCalendar** похож на компонент **DateTimePicker**, работающий в режиме ввода дат. Правда, в компоненте **MonthCalendar** предусмотрены некоторые дополнительные возможности: можно допустить множественный выбор дат в некотором диапазоне (свойство **MultiSelect**), можно указывать в календаре номера недель с начала года (свойство **WeekNumbers**), перестраивать календарь, задавая

первый день каждой недели (свойство **FirstDayOfWeek**) и т.п. Для некоторых офисных приложений все это достаточно удобно.

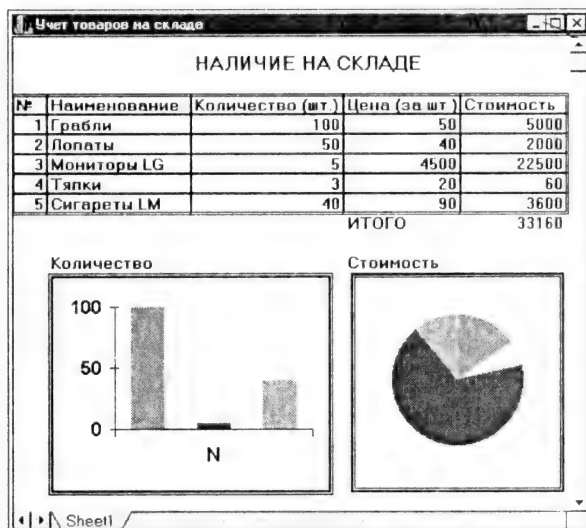
Компонент **Calendar** представляет собой менее красочный и более обыденно оформленный календарь на один месяц. Вместо свойства **Date** в нем предусмотрены отдельные свойства **Year** — год, **Month** — месяц, **Day** — день. Все это целые числа, с которыми иногда удобнее иметь дело, чем с типом **TDateTime**. Перед отображением на экране или в процессе проектирования надо задать значения **Month** и **Year**, чтобы компонент отобразил календарь на указанный месяц указанного года. Впрочем, если вам надо иметь календарь на текущий месяц, надо установить в **true** значение свойства **UseCurrentDate** (установлено по умолчанию). В этом случае по умолчанию будет показан календарь на текущий месяц с выделенным в нем текущим днем. Свойство **StartOfWeek** задает день, с которого начинается неделя. По умолчанию задано 0 — воскресенье, как это принято в западных календарях. Но для нас все-таки как-то привычнее начинать неделю с рабочего дня — понедельника. Так что желательно задать **StartOfWeek** = 1.

3.3.4 Страницы Excel — компонент F1Book

Очень интересным компонентом является **F1Book** на странице ActiveX. Этот компонент позволяет встроить в ваше приложение таблицы типа Excel (рис. 3.11), которые пользователь может заполнять соответствующими числами, а компонент будет производить по заданным формулам вычисления и тут же отображать их результаты в указанных ячейках. В таблицу можно встроить диаграммы и графики различных типов. И все изменения, вносимые пользователем в данные таблицы, немедленно будут отображаться в диаграммах. Таким образом вы можете включать в свое приложение различные бланки смет, счетов, ведомостей, с которыми будет работать пользователь, различные таблицы, производящие статистические или технические расчеты и т.п.

Рис. 3.11

Приложение с компонентом F1Book



Перенесите на форму компонент **F1Book** и щелкните на нем правой кнопкой мыши. Выберите из всплывшего меню команду **Workbook Designer**. Перед вами появится диалоговое окно проектирования, представленное на рис. 3.12. Те, кто знаком с программой Excel, могут увидеть, что это окно является несколько упрощен-

ным вариантом Excel. Проектирование таблицы производится фактически по тем же правилам, что и в Excel. Вы можете писать в ячейках необходимые надписи, задавая шрифт, его стиль, оформление. Можете записывать формулы. Так на рис. 3.11 и 3.12 последний столбец представляет собой стоимость соответствующего товара, являющуюся произведением его количества на его цену. А ячейка внизу таблицы суммирует стоимость всех товаров.

Рис. 3.12

Диалоговое окно проектирования компонента F1Book

№	Наименование	Количество (шт.)	Цена (за шт.)	Стоимость
1	Грабли	100	50	5000
2	Лопаты	50	40	2000
3	Мониторы LG	5	4500	22500
4	Тяпки	3	20	60
5	Сигареты LM	40	90	3600
ИТОГО				33160

Правая быстрая кнопка на рис. 3.12 позволяет ввести на страницу диаграммы и графики. Чтобы задать диаграмму, надо сначала выделить курсором в таблице данные, которые должны отображаться в диаграмме, затем нажать кнопку ввода диаграммы, после этого указать курсором рамку, в которой должна отображаться диаграмма. В результате вы попадете в диалоговое окно, в котором сможете выбрать тип диаграммы и необходимые ее атрибуты.

Рассказывать подробно о работе с окном проектирования компонента **F1Book** невозможно из-за ограничения на объем данной книги. Те, кто знаком с Excel, без труда смогут в этом окне ориентироваться. К тому же в нем имеется встроенная справка, вызываемая командой меню Help или клавишей F1.

Щелкнув правой кнопкой мыши на компоненте **F1Book**, вы можете выбрать еще одну команду — Properties. В появившемся при этом диалоговом окне вы можете, в частности, задать опции, определяющие, что будет видно или не видно в таблице при работе приложения: заголовки строк и столбцов (Row Heading и Column Heading), сетка (Gridlines), формулы вычислений (Formulas) и т.п.

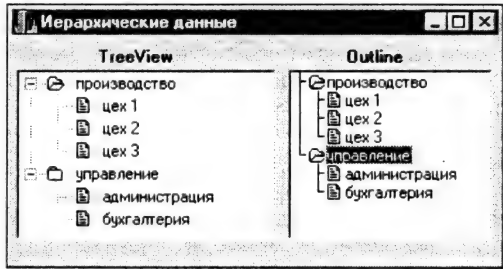
3.4 Компоненты отображения и ввода иных видов информации

3.4.1 Компоненты отображения иерархических данных — TreeView и Outline

Компоненты **TreeView** и **Outline** служат для отображения иерархических данных в виде дерева (см. пример на рис. 3.13), в котором пользователь может выбрать нужный ему узел или узлы. Иерархическая информация может быть самой разной: структура некоторого предприятия, структура документации учреждения, структура отчета и т.п. С каждым узлом дерева могут быть связаны некоторые данные.

Возможности компонента **TreeView** несколько шире, чем компонента **Outline**. К тому же **TreeView** — 32-разрядный компонент, а **Outline** — 16-разрядный. По-

Рис. 3.13
Примеры компонентов TreeView и Outline



этому **Outline** целесообразно использовать только в приложениях, предназначенных для работы в любых версиях Windows, включая Windows 3.x.

Основным свойством **TreeView**, содержащим информацию об узлах дерева, является **Items**. Доступ к информации об отдельных узлах осуществляется через свойство **Items[Index]**. Например, **TreeView1->Items->Item[0]** — это узел дерева с индексом 0 (первый узел дерева). Каждый узел является объектом типа **TTreeNode**, обладающим своими свойствами и методами.

Во время проектирования формирование дерева осуществляется в окне редактора узлов дерева, представленном на рис. 3.14. Это окно вызывается двойным щелчком на компоненте **TreeView** или нажатием кнопки с многоточием около свойства **Items** в окне Инспектора Объектов.

Кнопка **New Item** (новый узел) позволяет добавить в дерево новый узел. Он будет расположен на том же уровне, на котором расположен узел, выделенный курсором в момент щелчка на кнопке **New Item**.

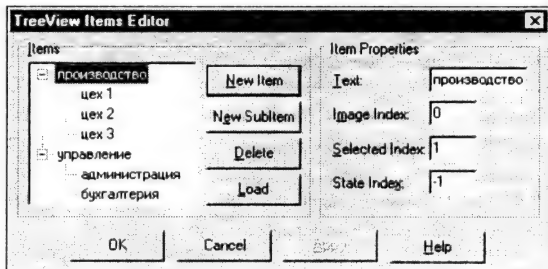
Кнопка **New SubItem** (новый дочерний узел) позволяет добавить в дерево дочерний узел. Он будет расположен на уровень ниже уровня того узла, который выделен курсором в момент щелчка на кнопке **New SubItem**.

Кнопка **Delete** (удалить) удаляет выделенный узел дерева. Кнопка **Load** позволяет загрузить структуру дерева из файла. Файл, хранящий структуру дерева — это обычный текстовый файл, содержащий тексты узлов. Уровни узлов обозначаются отступами. Например, файл дерева, изображенного на рис. 3.13 и 3.14, может иметь вид:

```
производство
  цех 1
  цех 2
  цех 3
управление
  администрация
  бухгалтерия
```

Для каждого нового узла дерева можно указать ряд свойств в панели **Item Properties** окна на рис. 3.14. Это прежде всего свойство **Text** — надпись, появляющаяся в дереве около данного узла. Свойства **Image Index** и **Selected Index** определяют индекс пиктограммы, отображаемой для узла, который соответственно не выделен и выделен пользователем в данный момент. Эти индексы соответствуют списку

Рис. 3.14
Окно редактора узлов дерева компонента TreeView



изображений, хранящихся в отдельном компоненте **ImageList** (см. раздел 3.9.2). Указание на этот компонент вы можете задать в свойстве **Images** компонента **TreeView**. Индексы начинаются с 0. Если вы укажете индекс -1 (значение по умолчанию), пиктограммы изображаться не будут. Последнее свойство — **State Index** в панели **Item Properties** позволяет добавить вторую пиктограмму в данный узел, не зависящую от состояния узла. Подобная пиктограмма может просто служить дополнительной характеристикой узла. Индекс, указываемый как **State Index**, соответствует списку изображений, хранящихся в отдельном компоненте **ImageList**, указанном в свойстве **StateImages** компонента **TreeView**.

Мы рассмотрели формирование дерева в процессе проектирования. Однако, дерево можно формировать или перестраивать и во время выполнения приложения. Для этого служит ряд методов объектов типа **TTreeNode**. Следующие методы позволяют вставлять в дерево новые узлы:

Add(TTreeNode* Node, const System::AnsiString S)	Добавляет новый узел с текстом S как последний узел уровня, на котором расположен Node
AddFirst(TTreeNode* Node, const System::AnsiString S)	Вставляет новый узел с текстом S как первый из узлов уровня, на котором находится Node . Индексы последующих узлов увеличиваются на 1
Insert(TTreeNode* Node, const System::AnsiString S)	Вставляет новый узел с текстом S сразу после узла Node на то же уровень. Индексы последующих узлов увеличиваются на 1
AddChild(TTreeNode* Node, const System::AnsiString S)	Добавляет узел с текстом S как последний дочерний узла Node
AddChildFirst(TTreeNode* Node, const System::AnsiString S)	Вставляет новый узел с текстом S как первый из дочерних узлов узла Node . Индексы последующих узлов увеличиваются на 1

Каждый из этих методов возвращает вставленный узел.

Ниже в качестве примера приведен код, формирующий то же дерево, которое вы можете видеть на рис. 3.13 и 3.14.

```
TreeView1->Items->Clear(); // очистка списка
// добавление корневого узла "производство" (индекс 0)
TreeView1->Items->Add(NULL, "производство");

/* добавление дочерних узлов "цех 1" — "цех 3"
(индексы 1 — 3) */
TreeView1->Items->AddChild(TreeView1->Items->Item[0], "цех 1");
TreeView1->Items->AddChild(TreeView1->Items->Item[0], "цех 2");
TreeView1->Items->AddChild(TreeView1->Items->Item[0], "цех 3");

/* добавление корневого узла "управление" после узла
"производство" (индекс 4) */
TreeView1->Items->Add(TreeView1->Items->Item[0], "управление");

/* добавление дочерних узлов "администрация" и "бухгалтерия"
узла "управление" */
TreeView1->Items->AddChild(
    TreeView1->Items->Item[4], "администрация");
TreeView1->Items->AddChild(
    TreeView1->Items->Item[4], "бухгалтерия");
```


Дерево может быть сколь угодно разветвленным. Например, следующие операторы добавляют дочерние узлы «бригада 1» и «бригада 2» в сформированный ранее узел «цех 1»:

```
TreeView1->Items->AddChild(TreeView1->Items->Item[1],
                           "бригада 1");
TreeView1->Items->AddChild(TreeView1->Items->Item[1],
                           "бригада 2");
```

Текст, связанный с некоторым узлом, можно найти с помощью его свойства **Text**. Например, **TreeView1->Items->Item[1]->Text** — это надпись «цех 1».

С каждым узлом может быть связан некоторый объект. Добавление таких узлов осуществляется методами **AddObject**, **AddObjectFirst**, **InsertObject**, **AddChildObject**, **AddChildObjectFirst**, аналогичными приведенным выше, но содержащими в качестве параметра еще указатель на объект:

AddObject (TTreeNode* Node, const System::AnsiString S, void * Ptr)	Добавляет новый узел с текстом S и объектом Ptr как последний узел уровня, на котором расположен Node
AddObjectFirst (TTreeNode* Node, const System::AnsiString S, void * Ptr)	Вставляет новый узел с текстом S и объектом Ptr как первый из узлов уровня, на котором находится Node . Индексы последующих узлов увеличиваются на 1
InsertObject (TTreeNode* Node, const System::AnsiString S, void * Ptr)	Вставляет новый узел с текстом S и объектом Ptr сразу после узла Node на то же уровень. Индексы последующих узлов увеличиваются на 1
AddChildObject (TTreeNode* Node, const System::AnsiString S, void * Ptr)	Добавляет узел с текстом S и объектом Ptr как последний дочерний узла Node
AddChildObjectFirst (TTreeNode* Node, const System::AnsiString S, void * Ptr)	Вставляет новый узел с текстом S и объектом Ptr как первый из дочерних узлов узла Node . Индексы последующих узлов увеличиваются на 1

Объект, связанный с некоторым узлом, можно найти с помощью его свойства **Data**. Например, **TreeView1->Items->Item[1]->Data**.

Для удаления узлов имеется два метода: **Clear(void)**, очищающий все дерево, и **Delete(TTreeNode* Node)**, удаляющий указанный узел **Node** и все его узлы — потомки. Например, оператор

```
TreeView1->Items->Clear();
```

удалит в нашем примере все узлы, а оператор

```
TreeView1->Items->Delete(TreeView1->Items->Item[1]);
```

удалит узел «цех 1» и его дочерние узлы (если они имеются).

При удалении узлов, связанных с объектами, сами эти объекты не удаляются.

Реорганизация дерева, связанная с созданием или удалением многих узлов, может вызывать неприятное мерцание изображения. Избежать этого можно с помощью методов **BeginUpdate** и **EndUpdate**. Первый из них запрещает перерисовку

дерева, а второй — разрешает. Таким образом, изменение структуры дерева может осуществляться по следующей схеме:

```
TreeView1->Items->BeginUpdate();
<операторы изменения дерева>
TreeView1->Items->EndUpdate();
```

Если метод **BeginUpdate** применен подряд несколько раз, то перерисовка дерева произойдет только после того, как столько же раз будет применен метод **EndUpdate**.

Среди свойств узлов следует отметить **Count** — число узлов, управляемых данным, т.е. дочерних узлов, их дочерних узлов и т.п. Если значение **Count** узла равно нулю, значит у узла нет дочерних узлов, т.е. он является листом дерева.

Вернемся к свойствам компонента **TreeView**. Важным свойством компонента **TreeView** является **Selected**. Это свойство указывает узел, который выделен пользователем. Пользуясь этим свойством можно запрограммировать операции, которые надо выполнить для выбранного пользователем узла. Если ни один узел не выбран, значение **Selected** равно **NULL**. При выделении пользователем нового узла происходят события **OnChanging** (перед изменением выделения) и **OnChanged** — после выделения. В обработчик события **OnChanging** передаются параметры **TTreeNode *Node** — узел, который выделен в данный момент, и **bool &AllowChange** — разрешение на перенос выделения. Если в обработчике задать **AllowChange = false**, то переключение выделения не произойдет. В обработчик события **OnChanged** передается только параметр **TTreeNode *Node** — выделенный узел. В этом обработчике можно предусмотреть действия, которые должны производиться при выделении узла.

У компонента **TreeView** имеется свойство **RightClickSelect**, разрешающее (при значении равном **true**) выделение узла щелчком как левой, так и правой кнопкой мыши. Но и в этом случае событие **OnChanged** наступает только при выделении узла левой кнопкой мыши.

Ряд событий компонента **TreeView** связан с развертыванием и свертыванием узлов. При развертывании узла происходят события **OnExpanding** (перед развертыванием) и **OnExpanded** (после развертывания). В обработчики обоих событий передается параметр **TTreeNode *Node** — развертываемый узел. Кроме того в обработчик **OnExpanding** передается параметр **bool &AllowExpansion**, который можно задать равным **false**, если желательно запретить развертывание. При свертывании узла происходят события **OnCollapsing** (перед свертыванием) и **OnCollapsed** (после свертывания). Так же, как и в событиях, связанных с развертыванием, в обработчики передается параметр **TTreeNode *Node** — свертываемый узел, а в обработчик **OnCollapsing** дополнительно передается параметр **bool &AllowCollapse**, разрешающий или запрещающий свертывание.

Свойство **ReadOnly** компонента **TreeView** позволяет запретить пользователю редактировать отображаемые данные. Если редактирование разрешено, то при редактировании возникают события **OnEditing** и **OnEdited**, аналогичные рассмотренным ранее (в обработчике **OnEditing** параметр **bool &AllowEdit** позволяет запретить редактирование).

Ряд свойств компонента **TreeView**: **ShowButtons**, **ShowLines**, **ShowRoot** отвечают за изображение дерева и позволяют отображать или убирать из него кнопки, показывающие раскрытия узла, линии, связывающие узлы, и корневой узел. Поэкспериментируйте с этими свойствами и увидите, как они влияют на изображение.

Свойство **SortType** позволяет автоматически сортировать ветви и узлы дерева. По умолчанию это свойство равно **stNone**, что означает, что дерево не сортируется. Если установить **SortType** равным **stText**, то узлы будут автоматически сортироваться по алфавиту. Возможно также проводить сортировку по связанным с узла-

ми объектам **Data** (значение **SortType** равно **stData**), одновременно по тексту и объектам **Data** (значение **SortType** равно **stBoth**) или любым иным способом. Для использования этих возможностей сортировки надо написать обработчик события **OnCompare**, в который передаются, в частности, параметры **TTreeNode *Node1** и **TTreeNode *Node2** — сравниваемые узлы, и параметр **int &Compare**, в который надо заносить результат сравнения: отрицательное число, если узел **Node1** должен располагаться ранее **Node2**, 0, если эти узлы считаются эквивалентными, и положительное число, если узел **Node1** должен располагаться в дереве после **Node2**. Например, следующий оператор в обработчике события **OnCompare** обеспечивает обратный алфавитный порядок расположения узлов:

```
Compare = - AnsiStrIComp(Node1->Text.c_str(),  
                        Node2->Text.c_str());
```

События **OnCompare** наступают после задания любого значения **SortType**, отличного от **stNone**, и при изменении пользователем свойств узла (например, при редактировании им надписи узла), если значение **SortType** не равно **stNone**. После сортировки первоначальная последовательность узлов в дереве теряется. Поэтому последующее задание **SortType = stNone** не восстанавливает начальное расположение узлов, но исключает дальнейшую генерацию событий **OnCompare**, т.е. автоматическую перестановку узлов, например, при редактировании их надписей. Если же требуется изменить характер сортировки или провести сортировку с учетом новых созданных узлов, то надо сначала задать значение **SortType = stNone**, а затем задать любое значение **SortType**, отличное от **stNone**. При этом будут сгенерированы новые обращения к обработчику событий **OnCompare**.

Имеются и другие возможности сортировки. Например, метод **AlphaSort(void)** обеспечивает алфавитную последовательность узлов независимо от значения **SortType**, но при отсутствии обработчика событий **OnCompare** (если обработчик есть, то при выполнении метода **AlphaSort** происходит обращение к этому обработчику). Отличие метода **AlphaSort** от задания значения **SortType = stText** заключается в том, что изменение надписей узлов приводит к автоматической пересортировке дерева только при **SortType = stText**.

Мы рассмотрели основные возможности компонента **TreeView**. Компонент **Outline** похож на него. Структура дерева тоже содержится в свойстве **Items** и доступ к отдельным узлам также осуществляется через этот индексный список узлов. Но индексы начинаются с 1. Например, **Outline1->Items[1] ->Text** — это текст узла дерева с индексом 1 (первого узла). Свойство **Items** имеет тип **TOutlineNode**. Его свойства и методы отличаются от свойств и методов типа узлов в **TreeView**. И заполняется структура дерева иначе: через свойство **Lines** типа **TStrings**. Редактор этого свойства можно вызвать, щелкнув на кнопке с многоточием около свойства **Lines** в окне Инспектора Объектов. Вы попадете в окно, в котором можете записать тексты всех узлов, делая отступы, чтобы выделить уровни узлов. Текст должен выглядеть так, как выше описывалось представление структуры в текстовом файле.

Пиктограммы, сопровождающие изображения узлов, задаются параметрами **PictureOpen** (пиктограмма развернутого узла), **PictureClosed** (пиктограмма свернутого узла), **PictureMinus** (пиктограмма символа «-» около развернутого узла, имеющего наследников), **PicturePlus** (пиктограмма символа «+» узла, имеющего наследников, но не развернутого), **PictureLeaf** (пиктограмма узла, не имеющего наследников — листа дерева). Основное отличие пиктограмм компонента **Outline** заключается в том, что они одинаковы для всех узлов одного типа, тогда как в **TreeView** можно задавать пиктограммы индивидуально для каждого узла.

Программно изменять структуру дерева можно с помощью методов **Add**, **AddObject** (добавление узла в дерево), **Insert**, **InsertObject** (вставка узла в задан-

ную позицию), **AddChild**, **AddChildObject** (вставка дочернего узла), **Delete** (удаление узла).

Индекс выделенного пользователем узла можно определить через свойство **SelectedItem**. Если **SelectedItem = 0**, значит ни один узел не выделен. Текст выделенного узла определяется свойством **Text**: например,

```
Outline1->Items[Outline1->SelectedItem]->Text
```

Впрочем, тот же самый текст даст и выражение

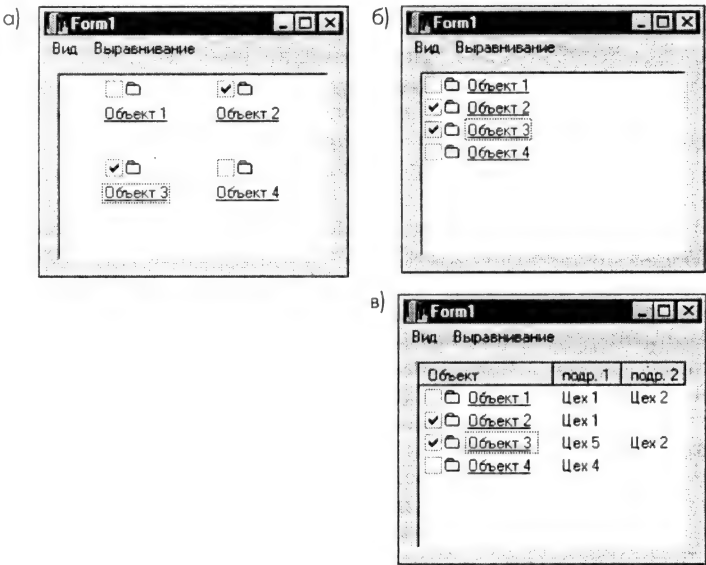
```
Outline1->Lines->Strings[Outline1->SelectedItem - 1]
```

3.4.2 Отображение информации в стиле папок Windows — компонент **ListView**

Компонент **ListView** позволяет отображать в стиле Windows данные в виде списков, таблиц, крупных и мелких пиктограмм. С подобным отображением все вы сталкиваетесь, раскрывая папки Windows.

Стиль отображения информации определяется свойством **ViewStyle**, которое может устанавливаться в процессе проектирования или программно во время выполнения. Свойство может принимать значения: **vsIcon** — крупные значки (см. рис. 3.15 а), **vsSmallIcon** — мелкие значки, **vsList** — список (см. рис. 3.15 б), **vsReport** — таблица (см. рис. 3.15 в). Что означает каждое из этих значений вы можете посмотреть не только на рис. 3.15, но и в любой папке Windows на рабочем столе.

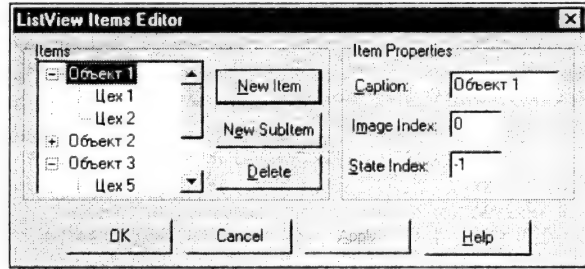
Рис. 3.15
Пример компонента **ListView** в режимах **vsIcon** (а), **vsList** (б) и **vsReport** (в)



Рассмотрим основные свойства и методы компонента **ListView**. Начните новое приложение и перенесите на него **ListView**. Основное свойство компонента **ListView**, описывающее состав отображаемой информации — **Items**. Во время проектирования оно может быть установлено специальным редактором (рис. 3.16), вызываемым щелчком на кнопке с многоточием рядом с этим свойством в окне Инспектора Объектов. Окно редактора похоже на окно, описанное для компонента **TreeView** (рис. 3.14). Точно так же в нем задаются новые узлы кнопкой **New Item** и дочерние узлы — кнопкой **New SubItem**. Только смысл дочерних узлов другой: это информация, которая появляется только в режиме **vsReport** — в виде таблицы.

Рис. 3.16

Окно редактора списка объектов компонента ListView



Задайте в редакторе некоторое множество элементов и дочерних узлов. Смысл их зависит от конкретной задачи. Это могут быть некоторые изделия и цеха, в которых они должны проходить обработку, или это могут быть товары и пункты, куда их надо отправить, и т.п. На рис. 3.16 указаны некоторые абстрактные объекты и цеха.

Для каждого узла задается свойство **Caption** — надпись, появляющаяся около пиктограммы. Для дочерних узлов это свойство соответствует надписи, появляющейся в ячейках таблицы в режиме **vsReport** (см. надписи вида «Цех ...» на рис. 3.15 в).

Свойство **Image Index** определяет индекс пиктограммы. Индекс соответствует спискам изображений, хранящимся в отдельных компонентах **ImageList** (см. раздел 3.9.2). Указания на эти компоненты вы можете задать в свойствах **LargeImages** для режима **vsIcon** и **SmallImages** для режимов **vsSmallIcon**, **vsList** и **vsReport**. Индексы начинаются с 0. Если вы укажете индекс -1 (значение по умолчанию), пиктограммы изображаться не будут.

Свойство **State Index** в панели **Item Properties** позволяет добавить вторую пиктограмму в данный объект. Подобная пиктограмма может просто служить дополнительной характеристикой объекта. Индекс, указываемый как **State Index**, соответствует списку изображений, хранящихся в отдельном компоненте **ImageList**, указанном в свойстве **StateImages** компонента **ListView**.

Остановимся пока на этих свойствах и построим приложение, предоставляющее пользователю возможность изменять вид списка в окне **ListView** и кроме того позволяющее при стилях **vsIcon** и **vsSmallIcon** перетаскивать пиктограммы мышью в любое место окна. Для реализации такого приложения нам потребуется компонент меню **MainMenu** (см. раздел 3.6.1). Кроме того для понимания некоторых приведенных ниже операторов надо представлять себе технологию перетаскивания **Drag&Drop**, с которой вы познакомитесь подробно в главе 4 в разделе 4.4.1. После изучения этого раздела вы сможете вернуться к приведенному ниже примеру и понять то, что в нем покажется вам неясным.

Для реализации примера надо сделать следующее:

- введите в приложение разделы меню Крупные значки (пусть его имя будет **MIcon**), Мелкие значки (имя **MSmallIcon**), Список (имя **MList**) и Таблица (имя **MReport**)
- установите во всех этих разделах одинаковый отличный от нуля индекс **GroupIndex** и свойства **RadioItem** в **true**
- один из разделов пометьте как **Checked** и в свойстве списка **ViewStyle** установите значение, соответствующее этому разделу
- напишите следующие обработчики щелчков для этих разделов:

```
void __fastcall TForm1::MIconClick(TObject *Sender)
{
    ListView1->ViewStyle = vsIcon;
    MIcon->Checked = true;
    ListView1->DragMode = dmAutomatic;
}
```

```

}
//-----
void __fastcall TForm1::MSmallIconClick(TObject *Sender)
{
    ListView1->ViewStyle = vsSmallIcon;
    MSmallIcon->Checked = true;
    ListView1->DragMode = dmAutomatic;
}
//-----
void __fastcall TForm1::MListClick(TObject *Sender)
{
    ListView1->ViewStyle = vsList;
    MList->Checked = true;
    ListView1->DragMode = dmManual;
}
//-----
void __fastcall TForm1::MReportClick(TObject *Sender)
{
    ListView1->ViewStyle = vsReport;
    MReport->Checked = true;
    ListView1->DragMode = dmManual;
}

```

Кроме того надо написать следующие обработчики событий **OnDragOver** и **OnDragDrop** компонента **ListView**:

```

void __fastcall TForm1::ListView1DragOver(TObject *Sender,
                                           TObject *Source, int X, int Y,
                                           TDragState State, bool &Accept)
{
    Accept = (Source = ListView1);
}
//-----
void __fastcall TForm1::ListView1DragDrop(TObject *Sender,
                                           TObject *Source, int X, int Y)
{
    ((TListView*) Sender)->Selected->Position = Point(X, Y);
}

```

Реализуйте этот пример. Вы увидите, что создали окно, имеющее много общего с обычными папками Windows.

Метод Arrange:

```
void __fastcall Arrange(TListArrangement Code);
```

позволяет упорядочить пиктограммы в режимах **vsIcon** и **vsSmallIcon**. Параметр **Code** определяет способ упорядочивания:

arAlignBottom	выравнивание вдоль нижнего края области
arAlignLeft	выравнивание вдоль левого края области
arAlignRight	выравнивание вдоль правого края области
arAlignTop	выравнивание вдоль верхнего края области
arDefault	выравнивание по умолчанию (вдоль верхнего края области)
arSnapToGrid	размещение каждой пиктограммы в ближайшем узле сетки

Вы можете ввести в свое тестовое приложение раздел Выравнивание и в обработчик щелчка на нем записать оператор

```
ListView1->Arrange(arAlignTop);
```

Тогда после перетаскивания элементов вы всегда сможете опять упорядочить их расположение, выбрав этот раздел меню.

Способ упорядочивания определяется соответствующим заданием свойства **SortType**, которое уже рассматривалось нами для компонента **TreeView**.

Свойство **Checkboxes**, установленное в **true**, определяет отображение индикатора с флажком около каждого элемента списка (см. рис. 3.15). Только учтите, что свойство срабатывает только в случае, если вы не установили описанное ранее свойство **StateImages**. Иначе говоря, около пиктограммы может появляться или индикатор, или дополнительная пиктограмма.

Индикаторы элементов можно устанавливать программно или их может изменять пользователь во время выполнения. Тогда узнать программно, установлен ли индикатор в некотором элементе **Items[i]**, можно проверкой его свойства **Checked**. Например:

```
for (int i=0; i < ListView1->Items->Count; i++)
    if (ListView1->Items->Item[i]->Checked)
        ShowMessage("Выбран элемент " +
            ListView1->Items->Item[i]->Caption);
```

Приведенный оператор проверяет все элементы и отображает сообщения о тех, в которых установлен индикатор. В реальном приложении, конечно, вместо такого сообщения должны предприниматься какие-то действия.

Свойства **HotTrack** и **HotTrackStyles** определяют появление выделения при перемещении курсора над элементом списка и стиль этого выделения. Свойство **HoverTime** задает в миллисекундах задержку появления такого выделения.

Свойство списка **Selected** определяет выделенный пользователем элемент списка. Этим можно воспользоваться для выполнения каких-то действий. Например, если требуются некие действия при двойном щелчке на каком-то элементе, то в обработчике события **OnDblClick** компонента **ListView** можно написать оператор:

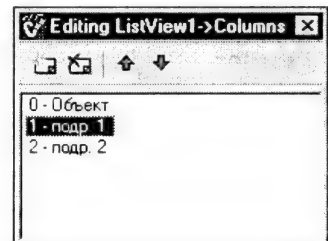
```
if (ListView1->Selected != NULL)
    ShowMessage(ListView1->Selected->Caption);
```

Оператор **if** проверяет, выделен ли какой-то элемент, т.е. произведен ли двойной щелчок на элементе, а не просто на пустом поле списка. Если щелчок на элементе, то с ним можно провести какие-то действия (в данном примере вместо этого просто отображается сообщение).

Свойство **Columns** определяет список заголовков таблицы в режиме **vsReport** (см. заголовки «Объект», «подр. 1», «подр. 2» на рис. 3.15 в) при свойстве **ShowColumnHeaders** (показать заголовки), установленном в **true**. Свойство **Columns** можно задать в процессе проектирования специальным редактором заголовков, вызываемом двойным щелчком на компоненте **ListView** или щелчком на кнопке с многоточием рядом со свойством **Columns** в окне Инспектора Объектов. В обоих случаях перед вами откроется окно редактора заголовков, представленное на рис. 3.17. Кнопка **Add New** (крайняя левая) позволяет добавить новую секцию в заголовков, кнопка **Delete Selected** (вторая слева) — удалить секцию, кнопки **Move Selected Up** и **Move Selected Down** (кнопки со стрелками) позволяют изменять последовательность секций.

Рис. 3.17

Окно редактора заголовков



После того, как вы добавили секцию и установили на ней курсор, в окне Инспектора Объектов появится множество свойств этого объекта. В свойстве **Caption** вы можете задать текст заголовка. В свойстве **ImageIndex** можете указать индекс пиктограммы, которая появится перед заголовком. Свойства **MinWidth** и **MaxWidth** определяют соответственно минимальную и максимальную ширину заголовка в пикселях. Только в этих пределах пользователь может изменять ширину заголовка курсором мыши. Значение ширины по умолчанию задается значением свойства **Width**. При изменении ширины секции во время выполнения генерируется событие **OnSectionResize**.

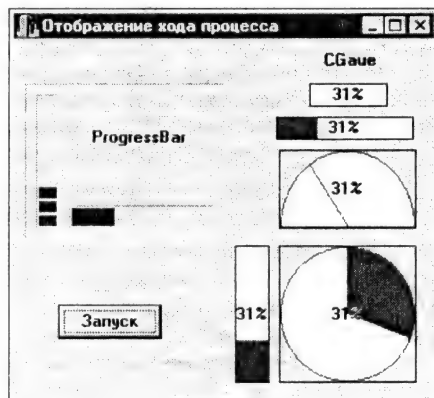
На этом мы завершим обсуждение свойств и методов **List View**. Более детальное рассмотрение вы можете найти в книге [2].

3.4.3 Отображение хода длительных процессов — компоненты **ProgressBar** и **CGauge**

Компоненты **ProgressBar** со страницы библиотеки Win32 и **CGauge** со страницы **Samples** предназначены для отображения хода процессов, занимающих заметное время, например, копирования больших файлов, настройку приложения, установку приложения на компьютере и т.п. Пример возможных вариантов отображения хода процесса компонентами **ProgressBar** и **CGauge** приведен на рис. 3.18.

Рис. 3.18

Пример отображения хода процесса компонентами **ProgressBar** и **CGauge**



Основные свойства этих компонентов очень схожи, различаясь только именами:

Свойство ProgressBar	Свойство CGauge	Описание
Max	MaxValue	Максимальное значение позиции (Position , Progress), которое соответствует завершению отображаемого процесса. По умолчанию задается в процентах — 100
Min	MinValue	Начальное значение позиции (Position , Progress), которое соответствует началу отображаемого процесса
Position	Progress	Позиция, которую можно задавать по мере протекания процесса, начиная со значения Min или MinValue в начале процесса, и кончая значением Max или MaxValue в конце. Если минимальное и максимальное значения выражены в процентах, то позиция — это процент завершенной части процесса

Свойство ProgressBar	Свойство CGauge	Описание
Smooth	—	Непрерывное (при значении true) или дискретное отображение процесса. На рис. 3.18 в горизонтальном компоненте ProgressBar задано Smooth = true , а в вертикальном — false
Step	—	Шаг приращения позиции, используемый в методе StepIt . Значение по умолчанию — 10
Orientation	—	Ориентация шкалы компонента: pbHorizontal — горизонтальная, pbVertical — вертикальная. Если задана ориентация pbVertical , то компонент надо вытянуть по вертикали (см. на рис. 3.18 компонент слева)
—	ForeColor	Цвет заполнения
—	ShowText	Текстовое отображение процента выполнения на фоне диаграммы
—	Kind	Тип диаграммы: gkHorizontalBar — горизонтальная полоса, gkVerticalBar — вертикальная полоса, gkPie — круговая диаграмма, gkNeedle — секторная диаграмма, gkText — отображение текстом

Отображение хода процесса можно осуществлять, задавая значение позиции — **Position** в **ProgressBar** или **Progress** в **CGauge**. Например, если полная длительность процесса характеризуется значением целой переменной **Count** (объем всех копируемых файлов, число настроек, количество циклов какого-то процесса), а выполненная часть — целой переменной **Current**, то задавать позицию диаграммы в случае, если используются значения минимальной и максимальной позиции по умолчанию (т.е. 0 и 100), можно операторами

```
ProgressBar1->Position = 100 * Current / Count;
```

или

```
CGauge1->Progress = 100 * Current / Count;
```

соответственно для **ProgressBar** и **CGauge**.

Можно поступать иначе: задать сначала значение максимальной величины равным **Count**, а затем в ходе процесса задавать позицию равной **Current**. Например:

```
CGauge1->MaxValue = Count;
...
CGauge1->Progress = Current;
```

Компонент **ProgressBar** имеет два метода, которыми тоже можно воспользоваться для отображения процесса: **StepBy(Delta: Integer)** — увеличение позиции на заданную величину **Delta**, и **StepIt** — увеличение позиции на один шаг, величина которого задается свойством **Step**. Ниже приведен пример использования метода **StepIt**:

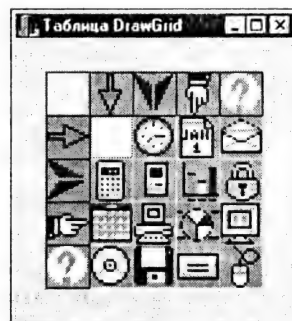
```
ProgressBar1->Max = Count;
ProgressBar1->Step = 1;
...
ProgressBar1->StepIt();
```

3.4.4 Таблицы изображений — компоненты DrawGrid и StringGrid

Компонент **DrawGrid** используется для создания в приложении таблицы, которая может содержать графические изображения (см. пример на рис. 3.19). Этот компонент подобен компоненту **StringGrid** (см. раздел 3.2.6), поскольку последний является производным от **DrawGrid**. Поэтому в **DrawGrid** присутствуют все свойства, методы, события компонента **StringGrid**, кроме относящихся к тексту, т.е. кроме свойств **Cells**, **Cols**, **Rows**, **Objects**. С этой точки зрения компонент **StringGrid** обладает существенно большими возможностями, чем **DrawGrid**, поскольку он может хранить в ячейках и изображения, и тексты. А если вы захотите внести текст в какие-то ячейки **DrawGrid**, то вам надо будет использовать для этого методы вывода текста на канву, что не очень удобно.

Рис. 3.19

Пример таблицы DrawGrid



Рассмотрим свойства компонентов **DrawGrid** и **StringGrid**, относящиеся к изображениям, поскольку свойства **StringGrid**, относящиеся к тексту, уже рассматривались в разделе 3.2.6.

Компоненты **DrawGrid** и **StringGrid** имеют канву **Canvas**, на которой можно размещать изображения методами, изложенными в главе 5. Имеется метод **CellRect**, который возвращает область канвы, отведенную под заданную ячейку. Этот метод определен как

```
Windows::TRect __fastcall CellRect(int ACol, int ARow);
```

где **ACol** и **ARow** — индексы столбца и строки, начинающиеся с 0, на пересечении которых расположена ячейка. Возвращаемая этой функцией область является областью канвы, в которой можно рисовать необходимое изображение. Например, оператор

```
DrawGrid1->Canvas->CopyRect(DrawGrid1->CellRect(1,1),  
    BitMap->Canvas, Rect(0,0, BitMap->Height, BitMap->Width));
```

копирует методом **CopyRect** (см. главу 5 раздел 5.1.5.2.) в ячейку (1,1) таблицы **DrawGrid1** изображение из компонента **BitMap**. Эта ячейка является второй слева и второй сверху в таблице, поскольку индексы начинаются с 0. Учтите, что если размеры ячейки меньше, чем размер копируемого изображения, то в ячейке появится только левая верхняя часть картинки.

Изображение на канве компонентов **DrawGrid** и **StringGrid**, как и на канве любого компонента, подвержено стиранию при перекрывании окна приложения другими окнами или, например, при сворачивании приложения. Поэтому необходимо принимать меры, описанные в главе 5 в разделе 5.1.7, чтобы с помощью обработчика событий **OnPaint** восстанавливать испорченное изображение. Это делает компонент **DrawGrid** не слишком удобным для использования.

Все свойства и события, позволяющие определить выбранную пользователем ячейку таблицы, были рассмотрены в разделе 3.2.6. Там же вы найдете описание

свойств, отвечающих за внешний вид и допустимость перестройки пользователем таблицы во время выполнения приложения.

3.4.5 Отображение форм — компонент Shape

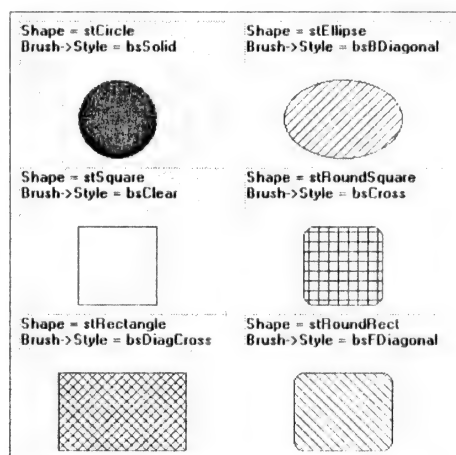
Компонент **Shape** представляет собой различные геометрические фигуры, соответствующим образом заштрихованные. Основное свойство этого компонента — **Shape** (форма), которое может принимать значения:

stRectangle	прямоугольник	stSquare	квадрат
stRoundRect	прямоугольник со скругленными углами	stRoundSquare	квадрат со скругленными углами
stEllipse	эллипс	stCircle	круг

Примеры этих форм показаны на рис. 3.20.

Рис. 3.20

Примеры компонента Shape



Другое существенное свойство компонента — **Brush** (кисть). Это свойство является объектом типа **TBrush**, имеющим ряд подсвойств, в частности: цвет (**Brush.Color**) и стиль (**Brush.Style**) заливки фигуры. Заливку при некоторых значениях **Style** вы можете видеть на рис. 3.20. Третье из специфических свойство компонента **Shape** — **Pen** (перо), определяющее стиль линий. Свойства **Brush** и **Pen** подробно рассмотрены в главе 5. Справочные данные об этих свойствах вы можете найти в главе 16.

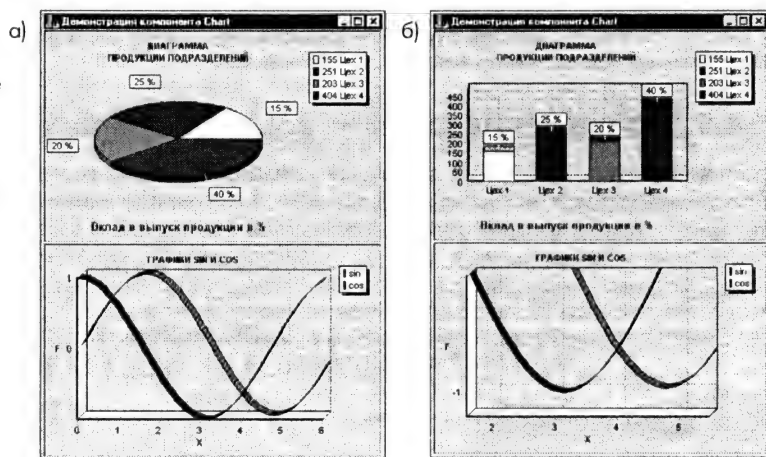
3.4.6 Графики и диаграммы — компонент Chart

Компонент **Chart** позволяет строить различные диаграммы и графики, которые выглядят очень эффектно (рис. 3.21). Компонент имеет множество свойств, методов, событий, так что если все их рассматривать, то этому пришлось бы посвятить целую главу. Поэтому ограничимся рассмотрением только основных характеристик **Chart**. А с остальными вы можете ознакомиться во встроенной справке C++Builder или просто опробовать их, экспериментируя с диаграммами.

Компонент **Chart** является контейнером объектов **Series** типа **TChartSeries** — серий данных, характеризующихся различными стилями отображения. Каждый компонент может включать несколько серий. Если вы хотите отображать график, то каждая серия будет соответствовать одной кривой на графике. Если вы хотите

Рис. 3.21

Пример приложения с диаграммами: начальное состояние (а) и состоянии при изменении типа диаграммы и увеличении фрагмента графика (б)



отображать диаграммы, то для некоторых видов диаграмм можно наложить друг на друга несколько различных серий, для других (например, для круговых диаграмм) это, вероятно, будет выглядеть некрасиво. Однако, и в этом случае вы можете задать для одного компонента **Chart** несколько серий одинаковых данных с разным типом диаграммы. Тогда, делая в каждый момент времени активной одну из них, вы можете предоставить пользователю выбор типа диаграммы, отображающей интересные его данные.

Разместите один или два (если захотите воспроизвести рис. 3.21) компонента **Chart** на форме и посмотрите открывшиеся в Инспекторе Объектов свойства. Приведем пояснения некоторых из них.

AllowPanning

Определяет возможность пользователя прокручивать наблюдаемую часть графика во время выполнения, нажимая правую кнопку мыши. Возможные значения: **pmNone** — прокрутка запрещена, **pmHorizontal**, **pmVertical** или **pmBoth** — разрешена соответственно прокрутка только в горизонтальном направлении, только в вертикальном или в обоих направлениях

AllowZoom

Позволяет пользователю изменять во время выполнения масштаб изображения, вырезая фрагменты диаграммы или графика курсором мыши (на рис. 3.21 б внизу показан момент просмотра фрагмента графика, целиком представленного на рис. 3.21 а)

Title

Определяет заголовок диаграммы

Foot

Определяет подпись под диаграммой. По умолчанию отсутствует. Текст подписи определяется подсвойством **Text**

Frame

Определяет рамку вокруг диаграммы

Legend

Легенда диаграммы — список обозначений

MarginLeft, MarginRight, MarginTop, MarginBottom

Значения левого, правого, верхнего и нижнего полей

BottomAxis, LeftAxis, RightAxis

Эти свойства определяют характеристики соответственно нижней, левой и правой осей. Задание этих свойств имеет смысл для графиков и некоторых типов диаграмм

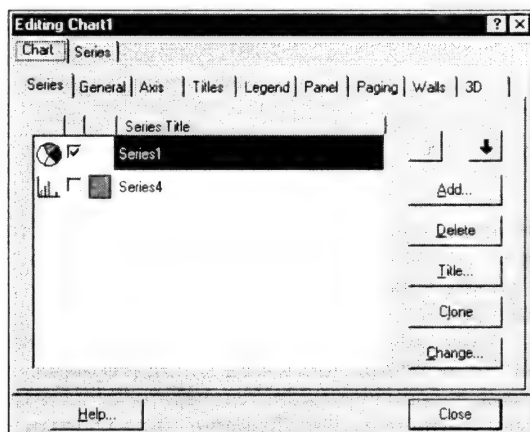
LeftWall, BottomWall, BackWall	Эти свойства определяют характеристики соответственно левой, нижней и задней граней области трехмерного отображения графика (см. рис. 3.21 а, нижний график)
SeriesList	Список серий данных, отображаемых в компоненте
View3d	Разрешает или запрещает трехмерное отображение диаграммы
View3DOptions	Характеристики трехмерного отображения
Chart3DPercent	Масштаб трехмерности (для рис. 3.21 это толщина диаграммы и ширина лент графика)

Рядом со многими из перечисленных свойств в Инспекторе Объектов расположены кнопки с многоточием, которые позволяют вызвать ту или иную страницу Редактора Диаграмм — многостраничного окна, позволяющего установить все свойства диаграмм. Вызов Редактора Диаграмм возможен также двойным щелчком на компоненте **Chart** или щелчком на нем правой кнопкой мыши и выбором команды Edit Chart во всплывшем меню.

Если вы хотите попробовать воспроизвести приложение, показанное на рис. 3.21, сделайте двойной щелчок на верхнем компоненте **Chart**. Вы попадете в окно Редактора Диаграмм (рис. 3.22) на страницу Chart, которая имеет несколько закладок. Прежде всего вас будет интересовать на ней закладка Series. Щелкните на кнопке Add — добавить серию. Вы попадете в окно (рис. 3.23), в котором вы можете выбрать тип диаграммы или графика. В данном случае выберите Pie — круговую диаграмму. Воспользовавшись закладкой Titles вы можете задать заголовок диаграммы, закладка Legend позволяет задать параметры отображения легенды диаграммы (списка обозначений) или вообще убрать ее с экрана, закладка Panel определяет вид панели, на которой отображается диаграмма, закладка 3D дает вам возможность изменить внешний вид вашей диаграммы: наклон, сдвиг, толщину и т.д.

Рис. 3.22

Редактор Диаграмм, страница Chart, закладка Series

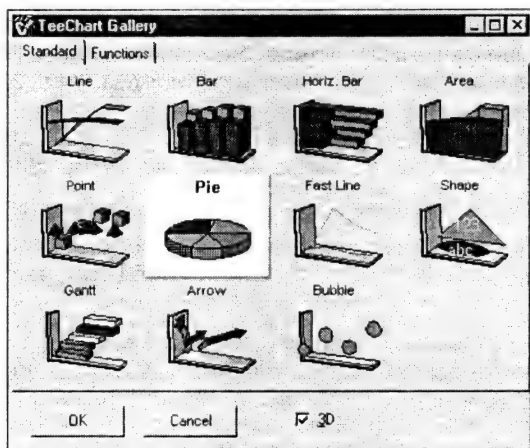


Когда вы работаете с Редактором Диаграмм и выбрали тип диаграммы, в компонентах **Chart** на вашей форме отображается ее вид с занесенными в нее условными данными (см. рис 3.24). Поэтому вы сразу можете наблюдать результат применения различных опций к вашему приложению, что очень удобно.

Страница Series, также имеющая ряд закладок, дает вам возможность выбрать дополнительные характеристики отображения серии. В частности, для круговой диаграммы на закладке Format полезно включить опцию Circled Pie, которая обеспечит при любом размере компонента Chart отображение диаграммы в виде круга. На закладке Marks кнопки группы Style определяют, что будет написано на ярлычках,

Рис. 3.23

Выбор типа диаграммы в Редакторе
Диаграмм

**Рис. 3.24**

Форма приложения рис. 3.21 с занесенными в нее
условными данными



относящихся к отдельным сегментам диаграммы: Value — значение, Percent — проценты, Label — названия данных и т.д. В примере рис. 3.21 включена кнопка Percent, а на закладке General установлен шаблон процентов, обеспечивающий отображение только целых значений.

Вы можете, если хотите, добавить на этот компонент Chart еще одну тождественную серию, нажав на закладке Series страницы Chart кнопку Clone, а затем для этой новой серии нажать кнопку Change (изменить) и выбрать другой тип диаграммы, например, Bar. Конечно, два разных типа диаграммы на одном рисунке будут выглядеть плохо. Но вы можете выключить индикатор этой новой серии на закладке Series, а потом предоставить пользователю выбрать тот или иной вид отображения диаграммы (ниже будет показано, как это делается).

Выйдите из Редактора Диаграмм, выделите в вашем приложении нижний компонент Chart и повторите для него задание свойств с помощью Редактора Диаграмм. В данном случае вам надо будет задать две серии, если хотите отображать на графике две кривые, и выбрать тип диаграммы Line. Поскольку речь идет о графиках, вы можете воспользоваться закладками Axis и Walls для задания координатных характеристик осей и трехмерных граней графика.

На этом проектирование внешнего вида приложения завершается. Осталось написать код, задающий данные, которые вы хотите отображать. Для тестового приложения давайте зададим в круговой диаграмме просто некоторые константные данные, а в графиках — функции синус и косинус.

Для задания отображаемых значений надо использовать методы серий **Series**. Остановимся только на трех основных методах.

Метод **Clear** очищает серию от занесенных ранее данных.

Метод **Add**:

```
long int Add(const double AValue,
             const String ALabel, TColor AColor);
```

позволяет добавить в диаграмму новую точку. Параметр **AValue** соответствует добавляемому значению, параметр **ALabel** — название, которое будет отображаться на диаграмме и в легенде, **AColor** — цвет. Параметр **ALabel** — не обязательный, его можно задать пустым: «».

Метод **AddXY**:

```
long int AddXY(const double AValue,
               const String ALabel, TColor AColor);
```

позволяет добавить новую точку в график функции. Параметры **AXValue** и **AYValue** соответствуют аргументу и функции. Параметры **ALabel** и **AColor** те же, что и в методе **Add**.

Таким образом, процедура, обеспечивающая загрузку данных в нашем примере, может иметь вид:

```
int A1=155;
int A2=251;
int A3=203;
int A4=404;
const Pi=3.14159;

Series1->Clear();
Series1->Add(A1, "Цех 1", clYellow);
Series1->Add(A2, "Цех 2", clBlue);
Series1->Add(A3, "Цех 3", clRed);
Series1->Add(A4, "Цех 4", clPurple);
Series2->Clear();
Series3->Clear();
for (int i = 0; i <= 100; i++)
{
    Series2->AddXY(0.02*Pi*i, sin(0.02*Pi*i), "", clRed);
    Series3->AddXY(0.02*Pi*i, cos(0.02*Pi*i), "", clBlue);
}
```

Эту процедуру можно включить в обработку щелчка какой-нибудь кнопки, в команду меню или просто в событие **OnCreate** формы. Операторы **Clear** нужны, если в процессе работы приложения вы собираетесь обновлять данные. Без этих операторов повторное выполнение методов **Add** и **AddXY** только добавит новые точки, не удалив прежние.

Если вы предусмотрели, например, для данных, отображаемых в диаграмме, две серии **Series1** и **Series4** разных видов — Pie и Bar, то можете ввести процедуру, изменяющую по требованию пользователя тип диаграммы. Эту процедуру можно ввести в событие **OnClick** какой-нибудь кнопки, в команду меню или, например, просто в обработку щелчка на компоненте **Chart**. Для того, чтобы загрузить данные в **Series4** и сделать эту диаграмму в первый момент невидимой, можно вставить в конце приведенной ранее процедуры операторы

```
Series4->Assign(Series1);
Series4->Active = false;
```

Первый из этих операторов переписывает данные, помещенные в **Series1**, в серию **Series4**. А второй оператор делает невидимой серию **Series4**. Смену типа диаграммы осуществляет процедура

```
Series1->Active = ! Series1->Active;
Series4->Active = ! Series4->Active;
```

На рис. 3.21 б вы можете видеть результат переключения пользователя на другой вид диаграммы.

3.4.7 Графики и диаграммы — компоненты Chartfx и Graph

В C++Builder 5 на странице ActiveX имеется несколько интересных компонентов, предназначенных для отображения диаграмм и графиков. Отметим, в частности, уже рассмотренный нами в разделе 3.3.4 компонент **F1Book**. На приводившемся в этом разделе рис. 3.11 показано, как можно использовать **F1Book** для отображения диаграмм и графиков данных, занесенных в таблицу.

Еще одним интересным компонентом на странице ActiveX является **Chartfx**. Он представляет собой законченный редактор диаграмм со встроенной инструментальной панелью (рис. 3.25). Нажимая кнопки инструментальной панели пользователь может задавать новые данные (самая правая кнопка на рис. 3.25 и выбор опции Data Editor), изменять тип диаграммы, сохранять диаграмму в файле с расширением **.chf** или загружать ее из аналогичного файла, копировать диаграмму в буфер обмена Clipboard и таким образом включать ее в другие документы (например, в документы Word) и т.п.

Рис. 3.25

Приложение на основе компонента Chartfx

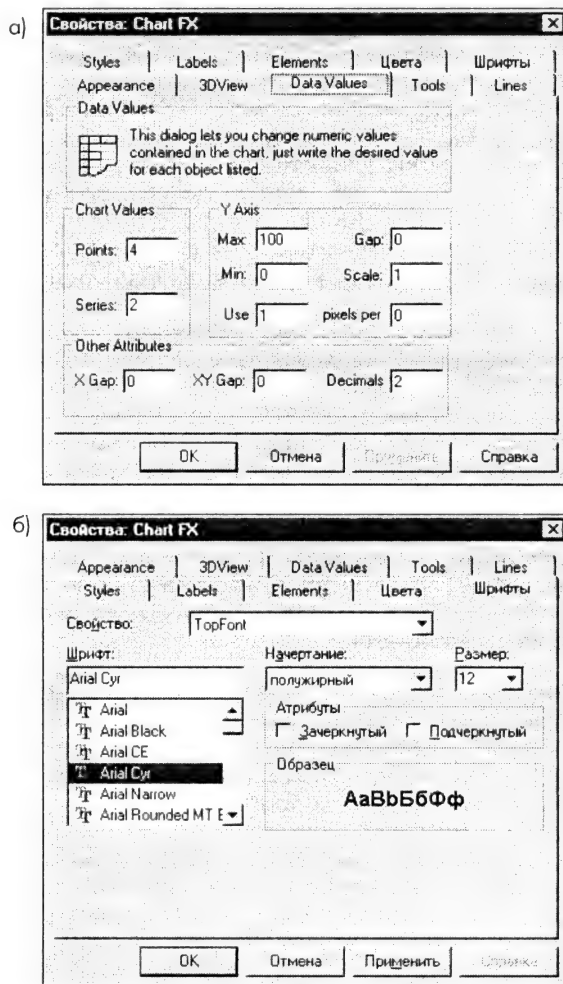


Как и в других компонентах ActiveX, доступ к свойствам **Chartfx** во время проектирования может осуществляться с помощью Инспектора Объектов или щелчком правой кнопки мыши и выбором из всплывшего меню команды Properties (Свойства). При выборе этой команды вы попадете в многостраничное диалоговое окно (см. рис. 3.26), позволяющее задать свойства компонента. Остановимся только на нескольких из них, которые вы можете задавать в этом диалоге, в Инспекторе Объектов или программно.

Свойство **Series** на странице Data Values (рис. 3.26 а) диалога (в Инспекторе Объектов это свойство названо **NSeries**) обозначает число серий данных (на рис. 3.25 равно 2). Свойство **Points** на той же странице диалога (в Инспекторе Объектов оно названо **NValues**) обозначает число значений по оси аргументов (на рис. 3.25 равно 4). Страница диалога Elements позволяет задать какие-то характерные уровни (опции Value — Text, на рис. 3.25 отмечен уровень 40 с текстом «Мини-

Рис. 3.26

Окно задания свойств компонента Chartfx: страницы Data Values (а) и Шрифты (б)



мально допустимый запас»), выделить цветом какие-то полосы уровней (опции From — To — Color, на рис. 3.25 выделены полосы 0 — 20 и 20 — 40), задать текст в строке состояния (опции ID — Width — Text). Прочие свойства позволяют задать тексты вверху диаграммы, внизу, слева, справа, задать координатные сетки и многое другое. Следует обратить внимание на выбор шрифтов на странице Шрифты (рис. 3.26 б). Шрифты, естественно, надо выбрать такие, которые содержат символы кириллицы. При выборе шрифтов надо сначала в выпадающем списке Свойство выбрать надпись, для которой указывается шрифт (например, **TopFont** — шрифт надписи над диаграммой), а затем в окнах Шрифт, Начертание, Размер установить атрибуты шрифта. Подобную процедуру надо повторить для всех надписей.

Теперь давайте коротко рассмотрим некоторые кнопки инструментальной панели компонента во время выполнения приложения (см. рис. 3.25). Первая и вторая слева быстрые кнопки обеспечивают соответственно чтение и сохранение диаграммы. Диаграмма сохраняется в файле с расширением **.chf** и может быть прочитана в последующих сеансах работы. Третья кнопка слева заносит диаграмму в буфер обмена Clipboard, откуда ее можно взять в каком-то другом приложении (например, в Word) и вставить в документ. Кнопки в центральной части панели позволяют изменять тип диаграммы или графика. Поэкспериментируйте с ними и

вы легко поймете, что они означают. Вторая справа группа кнопок позволяет вводить на диаграмме или графике координатную сетку. Правая группа кнопок обеспечивает задание надписей на изображении, выбор шрифта надписей и т.п. Главной из этих кнопок является крайняя правая. Она вызывает выпадающее меню, содержащее, в частности, раздел *Data Editor*. Если вы выберете этот раздел, вместо диаграммы вы увидите окно редактора данных, отображаемых на графике или в диаграмме. Сделав двойной щелчок на том или ином числе, вы можете изменить его. После того, как вы отредактировали данные, опять щелкните на второй справа кнопке инструментальной панели и снимите выделение с раздела *Data Editor*. Вы опять увидите диаграмму, отображающую введенные вами данные.

Более простым, но, возможно, более полезным на странице библиотеки ActiveX является компонент **Graph** (рис. 3.27). Он позволяет отображать данные в виде диаграмм различных типов. Настройку можно проводить так же, как и в других компонентах ActiveX, или в Инспекторе Объектов, или командой *Properties* из меню, всплывающего при щелчке правой кнопкой мыши на компоненте. В многостраничном диалоговом окне, всплывающем при выполнении этой команды, на странице **Graph** можно выбрать тип диаграммы или графика (опция **GraphType**) и число точек аргумента (**NumPoints**). На странице *Data* можно задать данные: **GraphData** — значения функции в различных точках, **XPosData** — значения аргументов в этих точках, **ColorData** — значения цветов в точках (для диаграмм). Остальные опции достаточно похожи на те, которые рассматривались для предыдущих компонентов.

Рис. 3.27

Приложение на основе компонента Graph



Во время выполнения приложения задать новые данные, отредактировать их, настроить график или диаграмму можно методом **BrowseProperties**. Например, оператор

```
Graph1->BrowseProperties();
```

вызовет то же многостраничное окно свойств, которое вы видите во время проектирования, выполняя команду *Properties* из всплывающего меню компонента. В этом окне можно задать или отредактировать всю отображаемую компонентом информацию. Изменять во время выполнения данные и способ их отображения можно также задавая значения свойствам компонента, большинство из которых вы можете видеть в процессе проектирования в окне Инспектора Объектов.

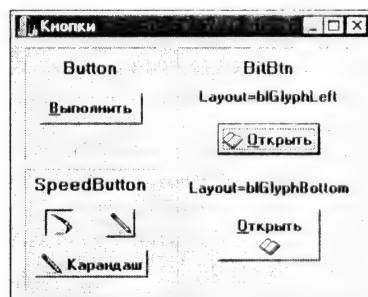
3.5 Кнопки, индикаторы, управляющие элементы

3.5.1 Управляющие кнопки Button и BitBtn

На рис. 3.28 показаны примеры кнопок **Button**, **BitBtn** и рассмотренной в следующем разделе кнопки **SpeedButton**. Простейшей и, пожалуй, наиболее часто используемой кнопкой является кнопка **Button** (на рис. 3.28 в верхнем левом углу формы), расположенная на странице библиотеки **Standard**. Реже используется кнопка **BitBtn** (на рис. 3.28 справа), отличающаяся, прежде всего, возможностью отобразить на ее поверхности изображение. Большинство свойств, методов и событий у этих видов кнопок одинаковы.

Рис. 3.28

Примеры кнопок



Основное с точки зрения внешнего вида свойство кнопки — **Caption** (надпись). В надписях кнопок можно предусматривать использование клавиш ускоренного доступа, выделяя для этого один из символов надписи. Перед символом, который должен соответствовать клавише ускоренного доступа, ставится символ амперсанта «&». Этот символ не появляется в надписи, а следующий за ним символ оказывается подчеркнутым. Тогда пользователь может вместо щелчка на кнопке нажать в любой момент клавишу **Alt** совместно с клавишей выделенного символа.

Например, если в вашем приложении имеется кнопка выполнения какой-то операции, вы можете задать ее свойство **Caption** равным «&Выполнить». На кнопке эта надпись будет иметь вид «Выполнить». И если пользователь нажмет клавиши **Alt-B**, то это будет эквивалентно щелчку на кнопке.

Основное событие любой кнопки — **OnClick**, возникающее при щелчке на ней. Именно в обработчике этого события записываются операторы, которые должны выполняться при щелчке пользователя на кнопке. Помимо этого есть еще ряд событий, связанных с различными манипуляциями клавишами и кнопками мыши. Эти события подробно рассмотрены в главе 4 разделе 4.3.

Свойство **Cancel**, если установить его в **true**, определяет, что нажатие пользователем клавиши **Esc** будет эквивалентно щелчку на данной кнопке. Это свойство целесообразно задавать равным **true** для кнопок **Отменить** в различных диалоговых окнах, чтобы можно было выйти из диалога, нажав на эту кнопку или нажав клавишу **Esc**.

Свойство **Default**, если его установить в **true**, определяет, что нажатие пользователем клавиши ввода **Enter** будет эквивалентно нажатию на данную кнопку, даже если данная кнопка в этот момент не находится в фокусе. Правда, это работает, если в фокусе находится какой-то оконный компонент. Если же в момент нажатия **Enter** в фокусе находится другая кнопка, то все-таки сработает именно кнопка в фокусе.

Еще одно свойство — **ModalResult** используется в модальных формах, которые будут рассмотрены в разделе 4.5.2. В обычных приложениях значение этого свойства должно быть равно **mrNone**.

Из методов, присущих кнопкам, имеет смысл отметить один — **Click**. Выполнение этого метода эквивалентно щелчку на кнопке, т.е. вызывает событие кнопки **OnClick**. Этим можно воспользоваться, чтобы продублировать какими-то другими действиями пользователя щелчок на кнопке. Пусть, например, вы хотите, чтобы при нажатии пользователем клавиши с символом «С» или «с» в любой момент работы с приложением выполнялись операции, предусмотренные в обработчике события **OnClick** кнопки **Button1**. Поскольку неизвестно, какой компонент будет находиться в фокусе в момент этого события, надо перехватить его на уровне формы. Такой перехват осуществляется, если установить свойство формы **KeyPreview** в **true** (см. раздел 4.3.2). Тогда в обработчике события формы **OnKeyPresss** можно написать оператор

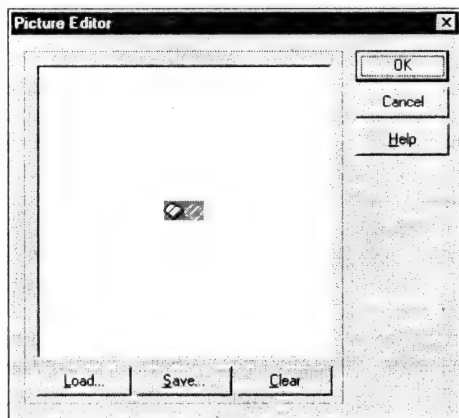
```
if ((Key=='C') || (Key=='c')) Button1->Click();
```

Если пользователь ввел символ «С» или «с», то в результате будет выполнен обработчик щелчка кнопки **Button1**.

Все сказанное выше в равной степени относится и к **Button**, и к **BitBtn**. Рассмотрим теперь особенности кнопки с пиктограммой **BitBtn**. Изображение на этой кнопке задается свойством **Glyph**. При нажатии кнопки с многоточием в строке свойства **Glyph** в Инспекторе Объектов вызывается окно, представленное на рис. 3.29. Нажав в нем кнопку **Load** вы перейдете в обычное окно открытия файла рисунка и можете выбрать файл битовой матрицы **.bmp**, содержащий желаемое изображение. В частности, с **C++Builder** поставляется большое количество изображений для кнопок. Они расположены в каталоге `\Images\Buttons`, а сам каталог `Images` в **C++Builder 5** обычно расположен в каталоге `...\Program Files\Borland\Borland Shared`.

Рис. 3.29

Окно редактора пиктограммы



После того, как вы выбрали изображение, нажмите **OK** и выбранное изображение появится на вашей кнопке левее надписи.

Файл изображения для кнопки может содержать до четырех изображений пиктограмм размера 16x16. Самое левое соответствует отжатой кнопке. Второе слева соответствует недоступной кнопке, когда ее свойство **Enabled** равно **false**. Третье слева изображение используется при нажатии пользователем на кнопку при ее включении. Четвертое слева изображение используется в кнопках с фиксацией **SpeedButton**, о которых будет сказано позднее, для изображения кнопки в нажатом состоянии. Большинство изображений для кнопок использует две пиктограммы. Число пиктограмм вы можете узнать из свойства кнопки **NumGlyphs**, которое после загрузки изображения покажет вам число пиктограмм в нем. Подробнее об изображениях на кнопках и о создании своих собственных пиктограмм см. в разделе 5.1.2.3.

Расположение изображения и надписи на кнопке определяется свойствами **Margin**, **Layout** и **Spacing**. Если свойство **Margin** равно -1 (значение по умолчанию), то изображение и надпись размещаются в центре кнопки. При этом положение изображения по отношению к надписи определяется свойством **Layout**, которое может принимать значения: **blGlyphLeft** (слева, это значение принято по умолчанию — см. верхнюю кнопку **BitBtn** на рис. 3.28), **blGlyphRight** (справа), **blGlyphTop** (вверху), **blGlyphBottom** (внизу — см. нижнюю кнопку **BitBtn** на рис. 3.28). Если же **Margin** > 0, то в зависимости от значения **Layout** изображение и надпись смещаются к той или иной кромке кнопки, отступая от нее на число пикселей, заданное значением **Margin**.

Свойство **Spacing** задает число пикселей, разделяющих изображение и надпись на поверхности кнопки. По умолчанию **Spacing** = 4. Если задать **Spacing** = 0, изображение и надпись будут размещены вплотную друг к другу. Если задать **Spacing** = -1, то текст появится посередине между изображением и краем кнопки.

Еще одно свойство **BitBtn** — свойство **Kind** определяет тип кнопки. По умолчанию значение этого свойства равно **bkCustom** — заказная. Но можно установить и множество других предопределенных типов: **bkOK**, **bkCancel**, **bkHelp**, **bkYes**, **bkNo**, **bkClose**, **bkAbort**, **bkRetry**, **bkIgnore**, **bkAll**. В этих типах уже сделаны соответствующие надписи, введены пиктограммы, заданы еще некоторые свойства. Обычно все-таки лучше ими не пользоваться. Во-первых, надписи все равно надо переводить на русский язык. Во-вторых, предопределенные рисунки обычно выбиваются из общего стиля конкретного приложения. И главное — предопределение некоторых свойств, не учтенных вами, может иногда приводить к странным результатам работы. Уж лучше использовать заказные кнопки и самому устанавливать в них все необходимые свойства.

3.5.2 Кнопка с фиксацией **SpeedButton**

Кнопки **SpeedButton** имеют возможность отображения пиктограмм и могут использоваться как обычные управляющие кнопки или как кнопки с фиксацией нажатого состояния (см. на рис. 3.28). Обычно они используются в качестве быстрых кнопок, дублирующих различные команды меню, и в инструментальных панелях, в которых требуется фиксация нажатого состояния.

У кнопок **SpeedButton**, как и у других кнопок, имеется свойство **Caption** — надпись, но в этих кнопках оно обычно оставляется пустым, так как вместо надписи используется пиктограмма. Изображение на кнопке задается свойством **Glyph** точно так же, как описано в разделе 3.5.1 для кнопок **BitBtn**. И точно так же свойство **NumGlyphs** определяет число используемых пиктограмм, свойства **Layout** и **Margin** определяют расположение изображения, а свойство **Spacing** — расстояние между изображением и надписью (если, конечно, вы все-таки хотите использовать надпись на кнопке).

Особенностью кнопок **SpeedButton** являются свойства **GroupIndex** (индекс группы), **AllowAllUp** (разрешение отжатого состояния всех кнопок группы) и **Down** (исходное состояние — нажатое). Если **GroupIndex** = 0, то кнопка ведет себя так же, как **Button** и **BitBtn**. При нажатии пользователем кнопки она погружается, а при отпускании возвращается в нормальное состояние. В этом случае свойства **AllowAllUp** и **Down** не влияют на поведение кнопки.

Если **GroupIndex** > 0 и **AllowAllUp** = **true**, то кнопка при щелчке пользователя на ней погружается и остается в нажатом состоянии. При повторном щелчке пользователя на кнопке она освобождается и переходит в нормальное состояние (именно для того, чтобы освобождение кнопки состоялось, необходимо задать **AllowAllUp** = **true**). Если свойство **Down** во время проектирования установлено равным **true**, то исходное состояние кнопки — нажатое.

Если есть несколько кнопок, имеющих одинаковое ненулевое значение **GroupIndex**, то они образуют группу взаимосвязанных кнопок из которых нажатой может быть только одна. Если одна кнопка находится в нажатом состоянии и пользователь щелкает на другой, то первая кнопка освобождается, а вторая фиксируется в нажатом состоянии. Поведение нажатой кнопки при щелчке на ней зависит от значения свойства **AllowAllUp**. Если оно равно **true**, то кнопка освободится, поскольку в этом случае возможно состояние, когда все кнопки группы отжаты. Если же **AllowAllUp** равно **false**, то щелчок на нажатой кнопке не приведет к изменению вида кнопки. Впрочем, и в этом случае, как и при любом щелчке на кнопке, возникает событие **OnClick**, которое может быть обработано.

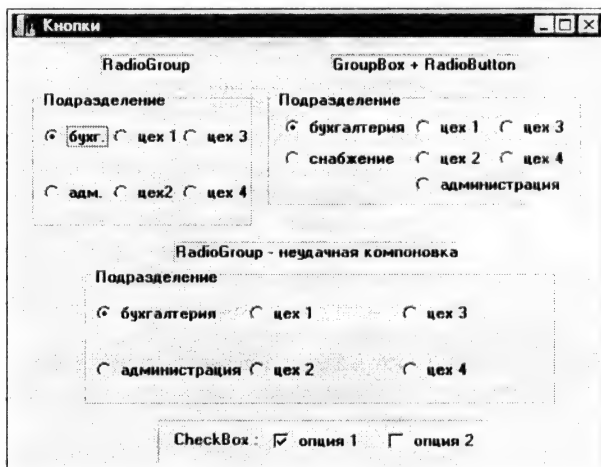
Состояние кнопки во время выполнения можно определить по значению свойства **Down**: если значение равно **true**, то кнопка нажата. Во время события **OnClick** значение **Down** уже равно тому состоянию, которое примет кнопка в результате щелчка на ней.

3.5.3 Группы радиокнопок — компоненты **RadioGroup**, **RadioButton** и **GroupBox**

Радиокнопки образуют группы взаимосвязанных индикаторов, из которых обычно может быть выбран только один. Они используются для выбора пользователем одной из нескольких взаимоисключающих альтернатив, например, отдела, в котором работает сотрудник, или пола сотрудника. Впрочем, радиокнопки могут использоваться не только для выбора, но и для отображения аналогичных данных. В этом случае управление кнопками осуществляется программно. Примеры размещения радиокнопок вы можете увидеть на рис. 3.30.

Рис. 3.30

Примеры радиокнопок и индикаторов



Начнем рассмотрение радиокнопок с компонента **RadioGroup** — панели группы радиокнопок. Это панель, которая может содержать регулярно расположенные столбцами и строками радиокнопки. Надпись в левом верхнем углу панели (см. рис. 3.30) определяется свойством **Caption**. А надписи кнопок и их количество определяются свойством **Items**, имеющим тип **TStrings**. Щелкнув на кнопке в многоугольнике около этого свойства в окне Инспектора Объектов, вы попадете в редактор списков строк, который уже рассматривался нами в разделе 3.2.4 (рис. 3.6). В нем вы можете занести надписи, которые хотите видеть около кнопок, по одной в строке. Сколько строчек вы запишете — столько и будет кнопок. Например, для компонента **RadioGroup** в нижней части формы рис. 3.30 свойство **Items** имеет вид:

```
бухгалтерия  
администрация  
цех 1  
цех 2  
цех 3  
цех 4
```

Кнопки, появившиеся в панели после задания значений **Items**, можно разместить в несколько столбцов (не более 17), задав свойство **Columns**. По умолчанию **Columns = 1**, т.е. кнопки размещаются друг под другом. На рис. 3.30 задано **Columns = 3**.

Определить, какую из кнопок выбрал пользователь, можно по свойству **ItemIndex**, которое показывает индекс выбранной кнопки. Индексы, как всегда в C++Builder, начинаются с 0. По умолчанию **ItemIndex = -1**, что означает отсутствие выбранной кнопки. Если вы хотите, чтобы в момент начала выполнения приложения какая-то из кнопок была выбрана (это практически всегда необходимо), то надо установить соответствующее значение **ItemIndex** во время проектирования. Если вы используете радиокнопки не для ввода, а для отображения данных, устанавливать значение **ItemIndex** можно программно во время выполнения приложения.

Компонент **RadioGroup** очень удобен, но не свободен от некоторых недостатков. Его хорошо использовать, если надписи кнопок имеют примерно одинаковую длину и если число кнопок в каждом столбце (при размещении их в нескольких столбцах) одинаково. Например, на рис. 3.30 группа радиокнопок в верхнем левом углу формы имеет нормальный вид. А группа аналогичных радиокнопок в нижней части формы выглядит плохо: она занимает слишком много места, которое пропадает впустую. Связано это с тем, что длина надписей у кнопок первого столбца превышает длину надписей у остальных кнопок. А **RadioGroup** при размещении кнопок ориентируется на надпись максимальной длины. Еще хуже выглядела бы эта группа, если бы число кнопок было, например, равно 5.

В подобных случаях желательно нерегулярное расположение кнопок. Такую возможность дают компоненты **RadioButton**, сгруппированные панелью **GroupBox** (в правом верхнем углу на рис. 3.30). Панель **GroupBox** выглядит на форме так же, как **RadioGroup**, и надпись в ее верхнем левом углу также определяется свойством **Caption**. Эта панель сама по себе пустая. Ее назначение — служить контейнером для других управляющих элементов, в частности, для радиокнопок **RadioButton**. Отдельная радиокнопка **RadioButton** особого смысла не имеет, хотя и может служить индикатором, включаемым и выключаемым пользователем. Но в качестве индикаторов обычно используются другие компоненты — **CheckBox**. А радиокнопки имеют смысл, когда они взаимодействуют друг с другом в группе. Эта группа и объединяется единым контейнером, обычно панелью **GroupBox**.

Рассмотрим свойства радиокнопки **RadioButton**. Свойство **Caption** содержит надпись, появляющуюся около кнопки. Значение свойства **Alignment** определяет, с какой стороны от кнопки появится надпись: **taLeftJustify** — слева, **taRightJustify** — справа (это значение принято по умолчанию). Свойство **Checked** определяет, выбрана данная кнопка пользователем, или нет. Поскольку в начале выполнения приложения обычно надо, чтобы одна из кнопок группы была выбрана по умолчанию, ее свойство **Checked** надо установить в **true** в процессе проектирования. Если вы поэкспериментируете, то заметите, что в **true** можно установить значение **Checked** только у одной кнопки из группы.

Размещение кнопок **RadioButton** в панели **GroupBox**, как можно видеть из рис. 3.30, дает большую свободу по сравнению с компонентом **RadioGroup** и позволяет разместить кнопки не регулярно.

Радиокнопки **RadioButton** могут размещаться не только в панели **GroupBox**, но и в любой панели другого типа, а также непосредственно на форме. Группа взаимосвязанных кнопок в этих случаях определяется тем оконным компонентом,

который содержит кнопки. В частности, для радиокнопок, размещенных непосредственно на форме, контейнером является сама форма. Таким образом, все кнопки, размещенных непосредственно на форме, работают как единая группа, т.е. только в одной из этих кнопок можно установить значение **Checked** в **true**.

3.5.4 Индикаторы **CheckBox** и **CheckBoxList**

Индикаторы с флажком **CheckBox** (см. в нижней части рис. 3.30 раздела 3.5.3) используются в приложениях в основном для того, чтобы пользователь мог включать и выключать какие-то опции, или для индикации состояния. При каждом щелчке пользователя на индикаторе его состояние изменяется, проходя в общем случае последовательно через три значения: выделение (появление черной галочки), промежуточное (серое окно индикатора и серая галочка) и не выделенное (пустое окно индикатора). Этим трем состояниям соответствуют три значения свойства компонента **State**: **cbChecked**, **cbGrayed**, **cbUnchecked**. Впрочем, эти три состояния допускаются только при значении другого свойства **AllowGrayed** равном **true**. Если же **AllowGrayed** = **false** (значение по умолчанию), то допускается только два состояния: выделенное и не выделенное. И **State**, и **AllowGrayed** можно устанавливать во время проектирования или программно во время выполнения.

Промежуточное состояние обычно используется, если индикатор применяется для отображения какой-то характеристики объекта. Например, если индикатор призван показать, какой регистр использовался при написании какого-то фрагмента текста, то в случае, если весь текст написан в верхнем регистре индикатор может принимать выделенное состояние, если в нижнем — не выделенное, а если использовались оба регистра — промежуточное.

Проверять состояние индикатора можно не только по значению **State**, но и по значению свойства **Checked**. Если **Checked** равно **true**, то индикатор выбран, т.е. **State** = **cbChecked**. Если **Checked** равно **false**, то **State** равно **cbUnchecked** или **cbGrayed**. Установка **Checked** в **true** во время проектирования или выполнения автоматически переключает **State** в **cbChecked**.

Как и в радиокнопке, в индикаторе **CheckBox** надпись задается свойством **Caption**, а ее размещение по отношению к индикатору — свойством **Alignment**.

Еще один компонент, имеющий индикаторы — список **CheckBoxList**. Это аналог рассмотренного в разделе 3.2.5 компонента **ListBox**, но около каждой строки списка имеется индикатор, состояние которого пользователь может изменять (см. рис. 3.7). Свойства, общие у **CheckBoxList** и **ListBox**, мы рассматривать не будем, так как все, характеризующее этот компонент как список, рассмотрено в разделе 3.2.5. А состояния индикаторов определяют два свойства: **State** и **Checked**. Оба эти свойства можно рассматривать как индексированные массивы, каждый элемент которого соответствует индексу строки. Общее количество элементов определяется свойством **Count** (только для чтения). Поскольку индексы начинаются с 0, то индекс последнего элемента равен **Count** — 1.

Свойства **State** и **Checked** можно устанавливать программно или читать, определяя установки пользователя. Например, операторы

```
CheckBoxList1->Checked[1] = true;  
CheckBoxList1->State[2] = cbGrayed;
```

устанавливают индикатор второй строки списка **CheckBoxList1** в состояние выбранного, а индикатор третьей строки — в промежуточное состояние (вспомним, что индексы начинаются с 0).

Оператор

```
for (int i = 0; i < CheckBoxList1->Items->Count; i++)  
    if (CheckBoxList1->Checked[i]) ...
```

проверяет состояние всех индикаторов списка, и для выбранных пользователем строк осуществляет какие-то действия (в приведенном операторе на месте этих действий просто поставлено многоточие).

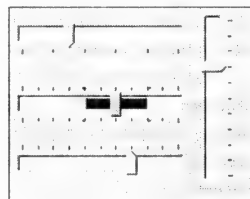
В компоненте **CheckListBox** имеется также событие **OnClickCheck**, возникающее при каждом изменении пользователем состояния индикатора. Его можно использовать для обработки результатов изменения.

3.5.5 Ползунки и полосы прокрутки — компоненты **TrackBar** и **ScrollBar**

Компонент **TrackBar** представляет собой элемент управления в виде ползунка, который пользователь может перемещать курсором мыши или клавишами во время выполнения. Таким образом, пользователь может управлять какими-то процессами: громкостью звука, размером изображения и т.п. На рис. 3.31 приведены различные формы отображения ползунка. Как видно из рисунка, он может располагаться горизонтально, вертикально, иметь шкалу с различных сторон, иметь какой-то выделенный диапазон шкалы.

Рис. 3.31

Различные варианты ползунков



Основное свойство компонента — **Position**. Это свойство можно задавать во время проектирования или программно во время выполнения. При перемещении пользователем ползунка можно прочитать значение **Position**, характеризующее позицию, в которую пользователь переместил ползунок. Для возможности такого чтения служит событие **OnChange**. В обработчике этого события можно прочитать значение **Position** и использовать его для управления каким-то компонентом.

Свойство **Position** — целое, значение которого может изменяться в пределах, задаваемых свойствами **Min** и **Max**. По умолчанию **Min** = 0, **Max** = 10, так что **Position** может принимать только 11 значений — от 0 до 10. Если задать большее значение **Max**, соответственно увеличится количество возможных значений **Position** в диапазоне **Min** — **Max**.

Свойство **Orientation** определяет ориентацию ползунка: **trHorizontal** — горизонтальная, **trVertical** — вертикальная.

Свойство **TickMarks** указывает размещение шкалы относительно компонента и может принимать значения: **tmBottomRight** — снизу или справа в зависимости от ориентации компонента (верхний и правый компоненты на рис. 3.31), **tmTopLeft** — сверху или слева в зависимости от ориентации компонента (нижний компонент на рис. 3.31), **tmBoth** — с обеих сторон (средний компонент на рис. 3.31).

Свойство **TickStyle** определяет способ изображения шкалы. Оно может принимать значения: **tsAuto** — автоматическая прорисовка шкалы, **tsNone** — отсутствие шкалы, **tsManual** — программное рисование шкалы с помощью метода **SetTick(Value: Integer)**, который помещает метку шкалы в позицию, соответствующую величине **Value**. Метки, соответствующие началу и концу шкалы автоматически размещаются и в случае **TickStyle** = **tsManual**.

При **TickStyle** = **tsAuto** частота меток шкалы определяется свойством **Frequency**. Это свойство задает, сколько возможных значений **Position** лежит между метками. Например, если **Frequency** = 2, то метки будут соответствовать толь-

ко каждому второму возможному значению позиции (такое значение **Frequency** задано в верхнем компоненте на рис. 3.31).

Свойства **LineSize** и **PageSize** определяют, насколько смещается ползунок, если пользователь управляет им с помощью соответственно клавиш со стрелками или клавишами **PageUp** и **PageDown**.

Свойства **SelStart** и **SelEnd** позволяют визуально выделить на шкале некоторый диапазон (см. средний компонент на рис. 3.31), который о чем-то говорит пользователю, например, рекомендуемый диапазон значений. При этом ничто не мешает пользователю выйти за пределы этого диапазона.

Похож на ползунок по своим функциям и компонент **ScrollBar**, хотя выглядит он иначе и предназначен по замыслу для других целей. Этот компонент представляет собой стандартную линейку прокрутки Windows. Однако, он может использоваться и для целей прокрутки (впрочем, многие оконные компоненты C++Builder имеют собственные полосы прокрутки), и для управления, подобно компоненту **TrackBar**.

Основные свойства **ScrollBar** — **Position**, **Min** и **Max** те же, что у компонента **TrackBar**. Свойство **Kind**, определяющее горизонтальное или вертикальное расположение полосы и принимающее соответственно значения **sbHorizontal** или **sbVertical**, аналогично свойству **Orientation** компонента **TrackBar**.

Имеются два свойства, отсутствующие у **TrackBar**: **SmallChange** и **LargeChange**. Они определяют соответственно «малый» сдвиг при щелчке на кнопке в конце полосы или нажатии клавиши со стрелкой, и «большой» сдвиг при перемещении на страницу щелчком рядом с бегунком или нажатием клавиш **PageUp** или **PageDown**.

Событие, соответствующее перемещению пользователем бегунка полосы прокрутки — **OnScroll**. В процедуру обработчика этого события передается по ссылке параметр **ScrollPos** — позиция бегунка, которую можно читать, но можно и изменять, и передается параметр **ScrollCode**, характеризующий вид перемещения бегунка. Этот параметр может иметь значения:

scLineUp, scLineDown	«Малый» сдвиг: перемещение соответственно вверх или налево и вниз или вправо после нажатия кнопки полосы прокрутки или клавиши со стрелкой
scPageUp, scPageDown	«Большой» сдвиг: перемещение на страницу щелчком рядом с бегунком или нажатием клавиш PageUp или PageDown
scPosition	Пользователь переместил и освободил бегунок
scTrack	Пользователь перемещает бегунок
scTop, scBottom	Бегунок перемещен соответственно в крайнюю верхнюю или левую позицию и в крайнюю нижнюю или правую позицию
scEndScroll	Окончание перемещения

В обработке события **ScrollPos** можно поместить операторы, перемещающие требуемую область формы или компонент, а можно поместить операторы, которые управляют некоторым компонентом, используя значение позиции бегунка **ScrollPos**.

3.5.6 Таймер — компонент Timer

Компонент **Timer** позволяет задавать в приложении интервалы времени. Таймер находит многочисленные применения: синхронизация мультимедиа, закрытие каких-то окон, с которыми пользователь долгое время не работает, включение хранителя экрана или закрытие связей с удаленным сервером при отсутствии действий пользователя, регулярный опрос каких-то источников информации, задание времени на ответ в обучающих программах — все это множество задач, в которых требуется задавать интервалы времени, решается с помощью таймера.

Таймер — невизуальный компонент, который может размещаться в любом месте формы. Он имеет два свойства, позволяющие им управлять: **Interval** — интервал времени в миллисекундах и **Enabled** — доступность. Свойство **Interval** задает период срабатывания таймера. Через заданный интервал времени после предыдущего срабатывания, или после программной установки свойства **Interval**, или после запуска приложения, если значение **Interval** установлено во время проектирования, таймер срабатывает, вызывая событие **OnTimer**. В обработчике этого события записываются необходимые операции.

Если задать **Interval = 0** или **Enabled = false**, то таймер перестает работать. Чтобы запустить отсчет времени надо или задать **Enabled = true**, если установлено положительное значение **Interval**, или задать положительное значение **Interval**, если **Enabled = false**.

Например, если требуется, чтобы через 5 секунд после запуска приложения закрылась форма — заставка, отображающая логотип приложения, на ней надо разместить таймер, задать в нем интервал **Interval = 5000**, а в обработчик события **OnTimer** вставить оператор **Close**, закрывающий окно формы.

Если необходимо в некоторой процедуре запустить таймер, который отсчитал бы заданный интервал, например, 5 секунд, после чего надо выполнить некоторые операции и отключить таймер, это можно сделать следующим образом. При проектировании таймер делается доступным (**Enabled = true**), но свойство **Interval** задается равным 0. Таймер не будет работать, пока в момент, когда нужно запустить таймер, не выполнится оператор

```
Timer1->Interval = 5000;
```

Через 5 секунд после этого наступит событие **OnTimer**. В его обработчике надо задать оператор

```
Timer1->Interval = 0;
```

который отключит таймер, после чего можно выполнять требуемые операции.

Другой эквивалентный способ решения задачи — использование свойства **Enabled**. В время проектирования задается значение **Interval = 5000** и значение **Enabled = false**. В момент, когда надо запустить таймер выполняется оператор

```
Timer1->Enabled = true;
```

В обработчик события **OnTimer**, которое наступит через 5 секунд после запуска таймера, можно вставить оператор

```
Timer1->Enabled = false;
```

который отключит таймер.

Таймер точно выдерживает заданные интервалы **Interval**, если они достаточно велики — сотни и тысячи миллисекунд. Если же задавать интервалы длительностью десятки или единицы миллисекунд, то реальные интервалы времени оказываются заметно больше вследствие различных накладных расходов, связанных с вызовами функций и иными вычислительными аспектами.

3.6 Компоненты — меню

3.6.1 Главное меню — компонент MainMenu

В C++Builder имеется два компонента, представляющие меню: **MainMenu** — главное меню, и **PopupMenu** — всплывающее меню. Оба компонента расположены на странице **Standard**. Эти компоненты имеют много общего. Начнем рассмотрение с компонента **MainMenu**.

Это невидимый компонент, т.е. место его размещения на форме в процессе проектирования не имеет никакого значения для пользователя — он все равно увидит не сам компонент, а только меню, сгенерированное им.

Обычно на форму помещается один компонент **MainMenu**. В этом случае его имя автоматически заносится в свойство формы **Menu**. Но можно поместить на форму и несколько компонентов **MainMenu** с разными наборами разделов, соответствующими различным режимам работы приложения. В этом случае во время проектирования свойству **Menu** формы присваивается ссылка на один из этих компонентов. А в процессе выполнения в нужные моменты это свойство можно изменять, меняя соответственно состав главного меню приложения.

Основное свойство компонента — **Items**. Его заполнение производится с помощью Конструктора Меню, вызываемого двойным щелчком на компоненте **MainMenu** или нажатием кнопки с многоточием рядом со свойством **Items** в окне Инспектора Объектов. В результате откроется окно, вид которого представлен на рис. 3.32. В этом окне вы можете спроектировать все меню. На рис. 3.33 показано в работе то меню, которое соответствует проектируемому на рис. 3.32.

Рис. 3.32

Окно Конструктора Меню

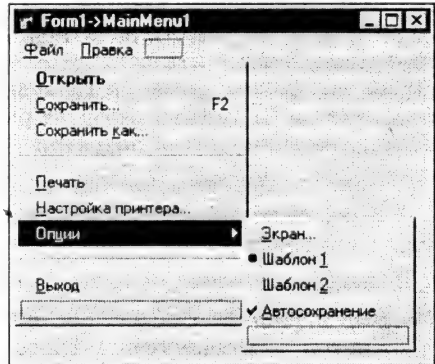
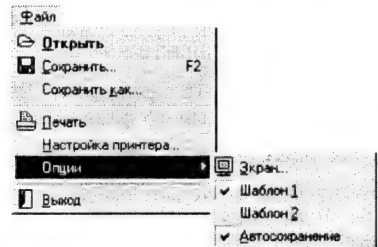


Рис. 3.33

Результат конструирования меню



При работе в конструкторе меню новые разделы можно вводить, помещая курсор в рамку из точек, обозначающую место расположения нового раздела (см. рис. 3.33). Если при этом раздел ввелся не на нужном вам месте, вы можете отбуксировать его мышью туда, куда вам надо. Другой путь ввода нового раздела — использование контекстного меню, всплывающего при щелчке правой кнопкой

мышь. Если вы предварительно выделите какой-то раздел меню и выберете из контекстного меню команду **Insert**, то рамка нового раздела вставится перед ранее выделенным. Из контекстного меню вы можете также выполнить команду **Create Submenu**, позволяющую ввести подменю в выделенный раздел (см. подменю раздела **Опции** на рис. 3.32, 3.33).

При выборе нового раздела вы увидите в Инспекторе Объектов множество свойств данного раздела. Дело в том, что каждый раздел меню, т.е. каждый элемент свойства **Items**, является объектом типа **TMenuItem**, обладающим своими свойствами, методами, событиями.

Свойство **Caption** обозначает надпись раздела. Заполнение этого свойства подчиняется тем же правилам, что и заполнение аналогичного свойства в кнопках (см. раздел 3.5.1), включая использование символа амперсанта для обозначения клавиш быстрого доступа. Если вы в качестве значения **Caption** очередного раздела введете символ минус «-», то вместо раздела в меню появится разделитель (см. на рис. 3.32 и 3.33 разделители после разделов **Сохранить как** и **Опции**).

Свойство **Name** задает имя объекта, соответствующего разделу меню. Очень полезно давать этим объектам осмысленные имена, так как иначе вы скоро запутаетесь в ничего не говорящих именах типа **N21**. Куда понятнее имена типа **MFile**, **MOpen**, **MSave** и т.п.

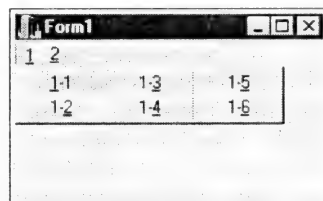
Свойство **Shortcut** определяет клавиши быстрого доступа к разделу меню — «горячие» клавиши, с помощью которых пользователь, даже не заходя в меню, может в любой момент вызвать выполнение процедуры, связанной с данным разделом. Чтобы определить клавиши быстрого доступа, надо открыть выпадающий список свойства **Shortcut** в окне Инспектора Объектов и выбрать из него нужную комбинацию клавиш. Эта комбинация появится в строке раздела меню (см. команду **Сохранить** на рис. 3.32, 3.33). В разделе 3.6.3 рассказано о некоторых дополнительных возможностях задания комбинаций горячих клавиш.

Свойство **Default** определяет, является ли данный раздел разделом по умолчанию своего подменю, т.е. разделом, выполняемым при двойном щелчке пользователя на родительском разделе. Подменю может содержать только один раздел по умолчанию, выделяемый жирным шрифтом (см. раздел **Открыть** на рис. 3.32, 3.33).

Свойство **Break** используется в длинных меню, чтобы разбить список разделов на несколько столбцов. Возможные значения **Break**: **mbNone** — отсутствие разбиения меню (это значение принято по умолчанию), **mbBarBreak** и **mbBreak** — в меню вводится новый столбец разделов, отделенный от предыдущего полосой (**mbBarBreak**) или пробелами (**mbBreak**). На рис. 3.34 показан пример, в котором в разделе 1-3 установлено значение **Break = mbBreak**, а в разделе 1-5 — **Break = mbBarBreak**.

Рис. 3.34

Пример меню с разбиением на столбцы



Свойство **Checked**, установленное в **true**, указывает, что в разделе меню будет отображаться маркер флажка, показывающий, что данный раздел выбран (см. на рис. 3.32, 3.33 раздел **Автосохранение**). Правда, сам по себе этот маркер не изменяется и в обработчик события **OnClick** такого раздела надо вставлять оператор типа

```
MAutoSave->Checked = ! MAutoSave->Checked;
```

(в приведенном операторе подразумевается, что раздел меню назван **MAutoSave**).

Еще одним свойством, позволяющим вводить маркеры в разделы меню, является **RadioItem**. Это свойство, установленное в **true**, определяет, что данный раздел должен работать в режиме радиокнопки совместно с другими разделами, имеющими то же значение свойства **GroupIndex**. По умолчанию значение **GroupIndex** равно 0. Но можно задать его большим нуля и тогда, если имеется несколько разделов с одинаковым значением **GroupIndex** и с **RadioItem = true**, то в них могут появляться маркеры флажков, причем только в одном из них (на рис. 3.32, 3.33 свойство **RadioItem** установлено в **true** в разделах Шаблон 1 и Шаблон 2, имеющих одинаковое значение **GroupIndex**). Если вы зададите программно в одном из этих разделов **Checked = true**, то в остальных разделах **Checked** автоматически сбросится в **false**. Впрочем, установка **Checked = true** лежит на программе; эта установка может выполняться аналогично приведенному выше оператору.

Описанные маркеры флажков в режиме радиокнопок и в обычном режиме используются для разделов меню, представляющих собой различные опции, взаимоисключающие или совместимые.

Для каждого раздела могут быть установлены во время проектирования или программно во время выполнения свойства **Enabled** (доступен) и **Visible** (видимый). Если установить **Enabled = false**, то раздел будет изображаться серой надписью и не будет реагировать на щелчок пользователя. Если же задать **Visible = false**, то раздел вообще не будет виден, а остальные разделы сомкнутся, заняв место невидимого. Свойства **Enabled** и **Visible** используются для того, чтобы изменить состав доступных пользователю разделов в зависимости от режима работы приложения.

Начиная с C++Builder 4 предусмотрена возможность ввода в разделы меню изображений. За это ответственны свойства разделов **Bitmap** и **ImageIndex**. Первое из них позволяет непосредственно ввести изображение в раздел, выбрав его из указанного вами файла. Второе позволяет указать индекс изображения, хранящегося во внешнем компоненте **ImageList** (см. раздел 3.9.2). Указание на этот компонент вы можете задать в свойстве **Images** компонента **MainMenu**. Индексы начинаются с 0. Если вы укажете индекс -1 (значение по умолчанию), изображения не будет.

Мы рассмотрели все основные свойства объектов, соответствующих разделам меню. Основное событие раздела — **OnClick**, возникающее при щелчке пользователя на разделе или при нажатии «горячих» клавиш быстрого доступа.

Рассмотрим теперь вопросы объединения главных меню вторичных форм с меню главной формы. Речь идет о приложениях с несколькими формами, в которых и главная, и вспомогательные формы имеют свои главные меню — компоненты **MainMenu**. Конечно, пользователю неудобно работать одновременно с несколькими окнами, каждое из которых имеет свое меню. Обычно надо, чтобы эти меню сливались в одно меню главной формы.

Приложения с несколькими формами могут быть двух видов: приложения с интерфейсом множества документов — так называемые MDI приложения, и обычные приложения с главной и вспомогательными формами. Типичными примерами приложений MDI являются программы Word и Excel. Подробнее это рассмотрено в разделах 4.1.2 и 4.5.4. Сейчас нас интересует только один вопрос: как объединяются меню различных форм. В MDI приложениях меню дочерних форм всегда объединяются с меню родительской формы. А в приложениях с несколькими формами наличие или отсутствие объединения определяется свойством **AutoMerge** компонентов **TMainMenu**. Если требуется, чтобы меню вторичных форм объединялись с меню главной формы, то в каждой такой вторичной форме надо установить **AutoMerge** в **true**. При этом свойство **AutoMerge** главной формы должно оставаться в **false**.

Способ объединения меню определяется свойством разделов **GroupIndex**. По умолчанию все разделы меню имеют одинаковое значение **GroupIndex**, равное

нулю. Если требуется объединение меню, то разделам надо задать неубывающие номера свойств **GroupIndex**. Тогда, если разделы встраиваемого меню имеют те же значения **GroupIndex**, что и какие-то разделы меню основной формы, то эти разделы заменяют соответствующие разделы основного меню. В противном случае разделы вспомогательного меню встраиваются между элементами основного меню в соответствии с номерами **GroupIndex**. Если встраиваемый раздел имеет **GroupIndex** меньший, чем любой из разделов основного меню, то разделы встраиваются в начало.

Пусть, например, в основной и вторичной формах структуры меню имеет следующие значения **GroupIndex**:

Форма 1	Форма 2
2 - 4	1 - 3
2 4	1 3
2 4	1

Тогда в момент, когда активизируется вторая форма, в первой появляется меню со структурой:

1 - 2 - 3 - 4
1 2 3 4
1 2 4

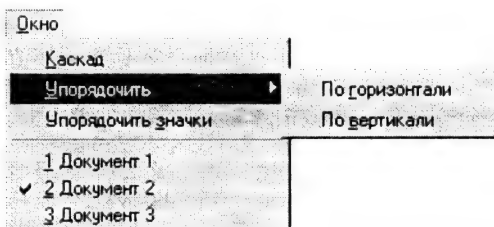
В этом примере отсутствовали разделы, имеющие в обеих формах одинаковые значения **GroupIndex**. Если бы такие были, то при активизации второй формы соответствующие разделы ее меню заменили бы аналогичные разделы первой формы.

Если в меню имеются разделы, работающие как радиокнопки, то нельзя забывать, что их взаимодействие также определяется свойствами **GroupIndex**.

Теперь остановимся на одном из вопросов, связанных с меню в упоминавшихся выше приложениях MDI. В них пользователь может открывать столько окон документов, сколько ему требуется. Обычно в подобных приложениях имеется меню Окно (см. рис. 3.35), которое содержит такие разделы, как Упорядочить. В конце меню идет обычно список открытых окон документов, в который заносятся названия открытых пользователем окон. Выбирая в этом списке, пользователь может переключаться между окнами документов.

Рис. 3.35

Меню «Окно» в приложении MDI со списком открытых документов



Для включения в меню раздела списка открытых окон, надо в свойстве **WindowMenu** главной формы приложения MDI указать имя меню, в конец которого должен помещаться список. Указывается именно имя меню, а не разделов выпадающего списка. Для примера рис. 3.35 должно быть указано имя элемента меню, соответствующего команде **Окно**.

В разделе 4.1.6 приведены требования, предъявляемые к меню приложений для Windows. Одним из требований является стандартизация меню и их разделов. Этому помогает команда *Save As Template* в контекстном меню, всплывающем при щелчке правой кнопкой мыши в окне Конструктора Меню. Эта команда вызывает диалог, представленный на рис. 3.36. В этом диалоге вы можете в верхнем окне указать описание (заголовок), под которым хотите сохранить ваше меню. Впоследствии в любом вашем новом приложении вы можете загрузить этот шаблон в меню, выбирая из всплывающего меню в окне Конструктора Меню команду *Insert From Template*.

Рис. 3.36

Окно сохранения шаблона разработанного меню



На этом мы пока закончим рассмотрение компонента **MainMenu**. В разделе 3.6.3 мы еще вернемся к нему и покажем на примере один из видов настройки меню в процессе выполнения приложения.

3.6.2 Контекстное всплывающее меню — компонент **PopupMenu**

Контекстное меню привязано к конкретным компонентам. Оно всплывает, если во время, когда данный компонент в фокусе, пользователь щелкнет правой кнопкой мыши. Обычно в контекстное меню включают те команды главного меню, которые в первую очередь могут потребоваться при работе с данным компонентом.

Контекстному меню соответствует компонент **PopupMenu**. Поскольку в приложении может быть несколько контекстных меню, то и компонентов **PopupMenu** может быть несколько. Оконные компоненты: панели, окна редактирования, а также метки и др. имеют свойство **PopupMenu**, которое по умолчанию пусто, но куда можно поместить имя того компонента **PopupMenu**, с которым будет связан данный компонент.

Формирование контекстного всплывающего меню производится с помощью Конструктора Меню, вызываемого двойным щелчком на **PopupMenu**, точно так же, как это делалось для главного меню. Обратим только внимание на возможность упрощения этой работы. Поскольку разделы контекстного меню обычно повторяют некоторые разделы уже сформированного главного меню, то можно обойтись копированием соответствующих разделов. Для этого, войдя в Конструктор Меню из компонента **PopupMenu**, щелкните правой кнопкой мыши и из всплывшего меню выберите команду *Select Menu* (выбрать меню). Вам будет предложено диалоговое окно, в котором вы можете перейти в главное меню. В нем вы можете выделить нужный вам раздел или разделы (при нажатой клавише Shift выделяются разделы в заданном диапазоне, при нажатой клавише Ctrl можно выделить совокупность разделов, не являющихся соседними). Затем выполните копирование их

в буфер обмена, нажав клавиши Ctrl-C. После этого опять щелкните правой кнопкой мыши, выберите команду Select Menu и вернитесь в контекстное меню. Укажите курсором место, в которое хотите вставить скопированные разделы, и нажмите клавиши чтения из буфера обмена — Ctrl-V. Разделы меню вместе со всеми их свойствами будут скопированы в создаваемое вами контекстное меню.

В остальном работа с **PopupMenu** не отличается от работы с **MainMenu**. Только не возникает вопросов объединения меню разных форм: контекстные меню не объединяются.

3.6.3 Горячие клавиши — компонент HotKey

Компонент **HotKey**, расположенный в библиотеке на странице Win32, является вспомогательным, обеспечивающим возможность задания самим пользователем горячих клавиш, определяющих быстрый доступ к разделам меню. К тому же этот компонент позволяет задать такие сочетания горячих клавиш, которые не предусмотрены в выпадающем списке свойства разделов меню **Shortcut**.

Компонент **HotKey** внешне выглядит как обычное окно редактирования **Edit**. Но если в него входит пользователь, то оно переводит нажимаемые им клавиши в тип **TShortcut**, хранящий комбинацию горячих клавиш. Например, если пользователь нажимает клавиши Ctrl-ф, то в окне **HotKey** появится текст «Ctrl + ф».

Основное свойство компонента — **HotKey**, равное по умолчанию комбинации клавиш Alt-A. Это свойство можно прочесть и присвоить свойству **Shortcut** какого-то раздела меню. Например, оператор

```
MOpen->Shortcut = HotKey1->HotKey;
```

присваивает разделу меню с именем **MOpen** комбинацию клавиш, заданную в компоненте **HotKey1**.

Свойство **Modifiers** указывает модификатор — вспомогательную клавишу, нажимаемую перед символьной. Это свойство является множеством, которое может включать значения **hkShift**, **hkCtrl**, **hkAlt**, **hkExt**, что соответствует клавишам Shift, Ctrl, Alt, Extra. По умолчанию **Modifiers** = [**hkAlt**]. Если вы хотите, например, задать вместо этого значения в качестве модификатора клавишу Ctrl, вы должны выполнить операторы:

```
HotKey1->Modifiers.Clear();  
HotKey1->Modifiers << hkCtrl;
```

Свойство **InvalidKeys** задает недопустимые клавиши или их комбинации. Это свойство является множеством, которое может включать значения **hcNone**, **hcShift**, **hcCtrl**, **hcAlt**, **hcShiftCtrl**, **hcShiftAlt**, **hcCtrlAlt**, **hcShiftCtrlAlt**, что соответствует отсутствию модификатора и клавишам Shift, Ctrl, Alt, Shift-Ctrl, Shift-Alt, Ctrl-Alt, Shift-Ctrl-Alt.

В заключение приведем пример использования компонента **HotKey** и настройки горячих клавиш меню в процессе выполнения приложения.

Пусть у вас есть главная форма приложения, содержащая компонент **MainMenu** и пусть вы хотите ввести команду настройки, позволяющую пользователю изменить установленные для разделов меню горячие клавиши. Для упрощения задачи будем считать, что меню, сконструированное в **MainMenu**:

- не каскадное (т.е. состоит только из двух уровней — заголовков меню и выпадающих списков разделов)
- в свойствах **Caption** разделов меню не использованы амперсаны
- в меню отсутствуют разделители

Эти предположения сделаны просто для того, чтобы упростить код.

Начните новое приложение, разместите на форме компонент **MainMenu** и сконструируйте с его помощью любое меню, удовлетворяющее перечисленным тре-

бованиям. Задайте каким-то из разделов меню быстрые клавиши. Один из разделов меню должен называться Настройка и при выборе его мы хотим предоставить пользователю вспомогательную форму для настройки быстрых клавиш.

Добавьте в приложение еще одну форму (команда File | New Form). Эта форма будет вспомогательной. В обработчик команды Настройка главной формы вставьте оператор

```
Form2->ShowModal();
```

Этот оператор покажет пользователю окно вспомогательной формы как модальное — т.е. пользователь не сможет вернуться в главную форму, пока не закроет вспомогательную. Чтобы компилятор понял этот оператор, надо в модуль главной формы **Unit1** вставить ссылку на заголовочный файл модуля вспомогательной формы **Unit2**. Можете вручную вставить директиву препроцессора

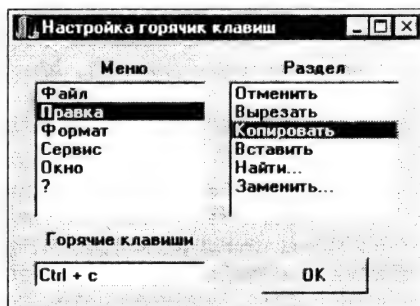
```
#include "Unit2.h"
```

А можете сделать это, перейдя в окне Редактора Кода в модуль **Unit1**, выполнив команду File | Use Unit и указав, что хотите связаться с модулем **Unit2**. Поскольку из модуля **Unit2** надо будет видеть меню модуля **Unit1**, то аналогичным образом введите и обратную связь — свяжите модуль **Unit2** с **Unit1**.

Теперь давайте спроектируем вспомогательную форму. Она может иметь вид, представленный на рис. 3.37. На ней расположено два списка **ListBox** (см. раздел 3.2.5): **ListBox1**, в котором отображаются заголовки меню, и **ListBox2**, в котором отображаются разделы меню, соответствующие выбранному заголовку. В нижней части формы расположен компонент **HotKey** и кнопка **Button**, которая фиксирует в меню сделанный пользователем выбор и закрывает форму. В компоненте **HotKey** надо стереть с помощью Инспектора Объектов свойство **HotKey**, которое содержит некоторое значение по умолчанию.

Рис. 3.37

Окно настройки горячих клавиш во время выполнения



```
void __fastcall TForm2::FormShow(TObject *Sender)
{
    /* Загрузка ListBox1 заголовками меню
       при событии OnShow формы Form2 */

    ListBox1->Clear();
    for(int i = 0; i < Form1->MainMenu->Items->Count; i++)
        ListBox1->Items->Add(
            Form1->MainMenu->Items->Items[i]->Caption);
    ListBox1->ItemIndex = 0;
    // Обращение к процедуре загрузки ListBox2
    ListBox1Click(Sender);
}
//_____

void __fastcall TForm2::ListBox1Click(TObject *Sender)
{

```

```

/* Загрузка ListBox2 заголовками разделов меню
   MainMenu1->Items->Items[ListBox1->ItemIndex],
   выделенного пользователем в ListBox1
   при событии OnShow формы Form2 */

ListBox2->Clear();
for(int i = 0; i < Form1->MainMenu1->Items->Items[
    ListBox1->ItemIndex]->Count; i++)
    ListBox2->Items->Add(Form1->MainMenu1->Items->Items[
        ListBox1->ItemIndex]->Items[i]->Caption);
ListBox2->ItemIndex = 0;
}
//-----
void __fastcall TForm2::ListBox2Click(TObject *Sender)
{
/* Занесение горячих клавиш выделенного в ListBox2 раздела
   в компонент HotKey1 */

HotKey1->HotKey = Form1->MainMenu1->Items->Items[
    ListBox1->ItemIndex]->Items[
    ListBox2->ItemIndex]->ShortCut;
}
//-----
void __fastcall TForm2::Button1Click(TObject *Sender)
{
/* Изменение горячих клавиш выбранного раздела меню
   и закрытие вспомогательной формы */

Form1->MainMenu1->Items->Items[
    ListBox1->ItemIndex]->Items[ListBox2->ItemIndex]->ShortCut
    = HotKey1->HotKey;

Close();
}

```

При событии **OnShow** формы **Form2** происходит загрузка списка **ListBox1** заголовками меню. Цикл загрузки перебирает индексы от 0 до **Form1->MainMenu1->Items->Count** — 1. Свойство **Count** равно числу элементов в меню.

При щелчке пользователя на списке **ListBox1** происходит загрузка списка **ListBox2**. При этом к соответствующим разделам меню получается доступ с помощью выражения **Form1->MainMenu1->Items->Items[ListBox1->ItemIndex]->Items[i]**. В этом выражении **Form1->MainMenu1->Items->Items[ListBox1->ItemIndex]** — элемент головного раздела меню, выбранного пользователем в **ListBox1**. Каждый такой раздел можно рассматривать как элемент массива меню и в то же время он сам является массивом разделов второго уровня. Поэтому его свойство **Items[i]** указывает на подраздел с индексом **i**.

При щелчке пользователя на списке **ListBox2** происходит загрузка компонента **HotKey1** символами горячих клавиш выбранного пользователем раздела. Если раздел не имеет горячих клавиш, то в окне **HotKey1** отображается текст «Нет». Далее пользователь может войти в окно **HotKey1** и нажать сочетание клавиш, которое он хочет назначить выбранному им разделу меню. Обработка щелчка на кнопке фиксирует это сочетание в разделе меню и закрывает вспомогательную форму.

Опробуйте это приложение в работе и вам станет яснее механизм работы с разделами меню и с быстрыми клавишами.

3.7 Панели и компоненты внешнего оформления

3.7.1 Общая характеристика

Панели являются контейнерами, служащими для объединения других управляющих элементов. Они могут выполнять как чисто декоративные функции, зрительно объединяя компоненты, связанные друг с другом по назначению, так и функции управления, организуя совместную работу своих дочерних компонентов. Примером этого была рассмотренная в разделе 3.5.3 организация работы группы радиокнопок.

В таблице 3.5 приведен перечень панелей, обслуживающих их компонентов и компонентов внешнего оформления, включенных в библиотеку C++Builder 5.

Таблица 3.5 Панели и обслуживающие их компоненты

Пиктограмма	Компонент	Страница	Описание
	GroupBox (групповое окно)	Standard	Является контейнером, объединяющим группу связанных органов управления, таких, как радиокнопки RadioButton , контрольные индикаторы Checkbox и т.д.
	Panel (панель)	Standard	Является контейнером для группирования органов управления и меньших контейнеров. Панель можно использовать также для построения полос состояния, инструментальных панелей, палитр инструментов.
	Bevel (рамка)	Additional	Используется для рисования прямоугольной рамки, изображенной как выступающая или утопленная.
	ScrollBox (окно с прокруткой)	Additional	Используется для создания зон отображения с прокруткой.
	Splitter (разделитель панелей)	Additional	Используется для создания в приложении панелей с изменяемыми пользователем размерами.
	HeaderControl (заголовок)	Win32	Позволяет создавать составные перемещаемые заголовки.
	Header (заголовок)	Win 3.1	Позволяет создавать составные перемещаемые заголовки, с меньшими возможностями, чем у HeaderControl .
	ControlBar (инструментальная панель)	Additional	Используется для размещения компонентов инструментальной панели.
	TabControl (страница с закладкой)	Win32	Позволяет добавлять закладки в стиле Windows, которые может выбирать пользователь.

Пиктограмма	Компонент	Страница	Описание
	PageControl (многостраничное окно)	Win32	Позволяет создавать страницы в стиле Windows, управляемые закладками или иными органами управления, для экономии места на рабочем столе.
	ToolBar (инструментальная панель)	Win32	Инструментальная панель для быстрого доступа к часто используемым функциям приложения.
	CoolBar (инструментальная перестраиваемая панель)	Win32	Контейнер инструментальной панели, размеры которой могут изменяться пользователем.
	PageScroller (прокрутка страниц)	Win32	Обеспечивает прокрутку больших окон, например, инструментальных панелей.
	TabSet (блокнот с закладками)	Win3.1	Используется для создания блокнота с закладками.
	TabbedNotebook (многостраничная форма)	Win3.1	Используется для создания многостраничных форм с закладками.
	Notebook (пачка страниц)	Win3.1	Используется для создания пачки страниц, может применяться совместно с TabSet .
	Frame (фрейм)	Standard	Используется как проектируемый в виде отдельного окна контейнер любых компонентов. Обладает возможностями наследования, может включаться в Депозитарий.

3.7.2 Панели общего назначения — компоненты Panel, GroupBox, Bevel, ScrollBox, Splitter

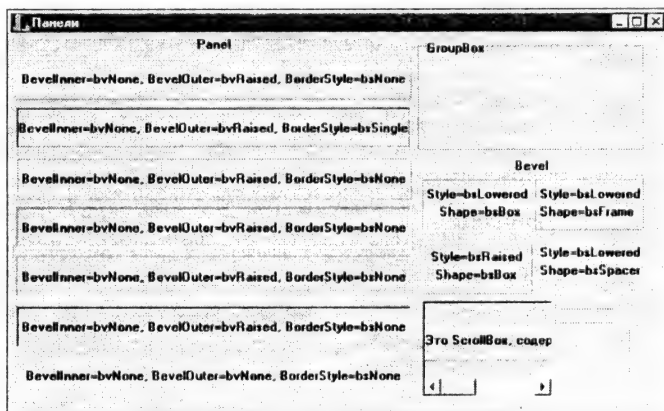
На рис. 3.38 приведен пример, демонстрирующий вид таких панелей, как **Panel**, **GroupBox**, **Bevel**, **ScrollBox**. В левой части формы размещены компоненты **Panel** с различными значениями параметров. С этих компонентов мы и начнем рассмотрение панелей.

Панели **Panel** используются наиболее широко. С их помощью конструируются различные элементы интерфейса (кнопки, окна редактирования, списки), функционально связанные друг с другом. Такая функциональная связь должна поддерживаться и зрительной связью — объединением соответствующих элементов в рамках одной панели. Панели **Panel** могут также использоваться для организации инструментальных панелей, полос состояния и т.п., хотя для этих целей имеются и специализированные компоненты, которые будут рассмотрены позднее.

Одним из назначений панелей является также группирование таких управляющих элементов, как **RadioButton** — радиокнопки (см. раздел 3.5.3). Все радиокнопки, расположенные на панели, работают как согласованная группа: в любой момент может быть выбрана только одна из них. Аналогично согласованной группой работают и расположенные на панели быстрые кнопки **SpeedButton** (см. раздел 3.5.2), если они имеют одинаковое значение свойства **GroupIndex**. В то же

Рис. 3.38

Пример панелей общего назначения



время **SpeedButton**, расположенные на разных панелях или на панели и форме, не образуют связанной группы даже при одинаковом значении **GroupIndex**.

Методика использования панелей подробно рассмотрена в главе 4 в разделе 4.2. Поэтому здесь стоит обратить внимание только на некоторые свойства компонентов **Panel**. Внешний вид панели **Panel** определяется совокупностью параметров **BevelInner** — стиль внутренней части панели, **BevelOuter** — стиль внешней части панели, **BevelWidth** — ширина внешней части панели, **BorderStyle** — стиль бордюра, **BorderWidth** — ширина бордюра. Результат сочетания некоторых значений этих параметров показан на рис. 3.38. Верхняя панель соответствует значениям параметров по умолчанию. Нижняя панель соответствует случаю, когда не определен стиль ни одной из областей панели (значения всех параметров равны **None**). В этом случае сама панель никак не выделяется на форме. Видна только надпись на ней (свойство **Caption**), если надпись задана, и, конечно, видны те компоненты, которые размещаются на панели.

Вопросы задания и форматирования надписи панели были рассмотрены в разделе 3.2.2, вопросы размещения панелей на форме и автоматического изменения их размеров рассмотрены в разделе 4.2. Там же рассмотрена методика построения панелей, размеры которых могут перестраиваться пользователем. В библиотеке **C++Builder** имеется специальный компонент — **Splitter**, который позволяет легко осуществить это. Его свойства **Beveled** и **ResizeStyle**, определяющие вид разделителя, и свойство **MinSize**, ограничивающее минимальный размер панелей по обе стороны от разделителя, рассмотрены в разделе 4.2.3.

Панель **GroupBox** не имеет таких широких возможностей задания различных стилей оформления, как **Panel**. Но она имеет встроенную рамку с надписью (см. рис. 3.38), которая обычно используется для выделения на форме группы функционально объединенных компонентов. Никаких особых свойств, отличных от уже рассмотренных, панель **GroupBox** не имеет.

Компонент **Bevel** формально не является панелью, он не может служить контейнером для компонентов. Например, с помощью **Bevel** нельзя сгруппировать радиокнопки. Однако, чисто зрительно компонент **Bevel** может использоваться как подобие панели. На рис. 3.38 в правой нижней части окна представлены различные варианты оформления **Bevel**.

Стиль отображения **Bevel** определяется свойством **Style**, которое может принимать значения **bsLowered** — утопленный, и **bsRaised** — приподнятый. А контур компонента определяется свойством **Shape**, которое может принимать значения: **bsBox** — прямоугольник, **bsFrame** — рамка, **bsSpacer** — пунктирная рамка, **bsTopLine**, **bsBottomLine**, **bsLeftLine**, **bsRightLine** — соответственной верхней, нижней, левой и правой линии. В зависимости от значения **Style** линии могут

быть утопленными или выступающими. Все перечисленные варианты приведены на рис. 3.38.

Остановимся теперь на компоненте **ScrollBar** — панели с прокруткой. Этот компонент предназначен для создания области, в которой могут размещаться компоненты, занимающие площадь большую, чем сам **ScrollBar**. Например, компонент **ScrollBar** можно использовать для размещения длинных текстовых строк или больших инструментальных панелей, которые исходя из соображений экономии площади окна нецелесообразно отображать целиком. В примере рис. 3.38 в **ScrollBar** помещена панель с надписью: «Это ScrollBox, содержащая панель с длинной надписью». В пределах **ScrollBar** видна только часть этой панели. Если размеры **ScrollBar** меньше, чем размещенные компоненты, то появляются полосы прокрутки, которые позволяют пользователю перемещаться по всем размещенным в **ScrollBar** компонентам.

Разместить в пределах небольшой области **ScrollBar** большие компоненты или много компонентов, занимающих в сумме большую площадь, можно в процессе проектирования, например, с помощью такого приема. Увеличьте временно размер **ScrollBar** так, чтобы в этом компоненте поместилось все, что вы хотите разместить. Проведите необходимое размещение. А затем сократите размеры **ScrollBar** до требуемых.

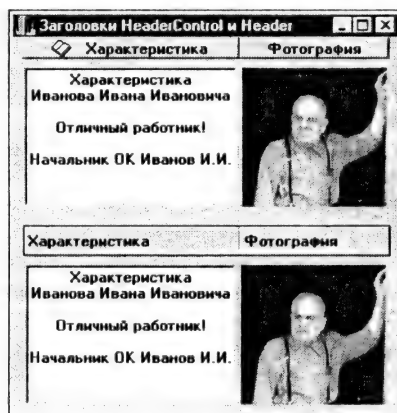
Свойство **BorderStyle** определяет стиль рамки компонента **ScrollBar**. Свойство **AutoScroll** позволяет задать автоматическое появление необходимых полос прокрутки, если размер размещенных компонентов превышает размер области по горизонтали, вертикали или в обоих измерениях. Если по каким-то соображениям это нежелательно, вы можете сами управлять появлением горизонтальной и вертикальной полос с помощью свойств **HorzScrollBar** и **VertScrollBar** соответственно. Но в этом случае вы должны сами задавать ряд свойств полосы прокрутки и, прежде всего, **Range** — размер в пикселях прокручиваемой области. Значение перемещения при однократном нажатии пользователем кнопки прокрутки может рассчитываться компонентом автоматически исходя из размеров области и окна, если свойство полосы прокрутки **Smooth** установлено в **true**. В противном случае вы должны задать величину единичного перемещения в свойстве **Increment**.

3.7.3 Заголовки — компоненты HeaderControl и Header

Компоненты заголовков **HeaderControl** и **Header** являются компонентами, с помощью которых можно управлять размещением расположенных под ними панелей. Заголовок состоит из ряда секций (см. рис. 3.39), причем пользователь во время выполнения приложения может изменять ширину отдельных секций с помощью мыши.

Рис. 3.39

Пример использования заголовков



По умолчанию свойство **Align** в **HeaderControl** задано равным **alTop**, что обеспечивает размещение компонента вверху окна формы. Но это свойство можно изменить, например, на **alNone** и разместить компонент в любом необходимом месте.

Основное свойство компонента **HeaderControl** — **Sections**. Оно является списком объектов типа **THeaderSection**, каждый из которых описывает одну секцию заголовка. Свойство **Sections** можно задать во время проектирования, нажав кнопку с многоточием рядом с этим свойством в Инспекторе Объектов или просто сделав двойной щелчок на компоненте **HeaderControl**. В обоих случаях перед вами откроется окно редактора заголовков, подобное рассмотренному ранее в разделе 3.4.2 и представленному на рис. 3.17. Левая быстрая кнопка позволяет добавить новую секцию в заголовок. Следующая быстрая кнопка позволяет удалить секцию. Кнопки со стрелкой позволяют изменять последовательность секций.

После того, как вы добавили секцию и установили на ней курсор, в окне Инспектора Объектов появится множество свойств этого объекта. В свойстве **Text** вы можете задать текст заголовка. Свойства **MinWidth** и **MaxWidth** определяют соответственно минимальную и максимальную ширину секции в пикселях. Только в этих пределах пользователь может изменять ширину секции курсором мыши. Значение ширины по умолчанию задается значением свойства **Width**. При изменении ширины секции во время выполнения генерируется событие **OnSectionResize**. В обработчик этого события надо вставить операторы, синхронно изменяющие ширину того, заголовком чего является секция: это может быть какая-то панель, таблица, изображение и т.п.

Свойство **AllowClick**, равное по умолчанию **true**, определяет поведение секции как кнопки при щелчке пользователя на ней. В этом случае при щелчке генерируется событие **OnSectionClick** компонента **HeaderControl**, в обработчик которого и надо вставить операторы, выполняющие необходимые действия.

Свойство **Style** может иметь значение **hsText** — в этом случае в заголовке отображается значение свойства **Text**, или **hsOwnerDraw** — в этом случае отображается то, что рисуется непосредственно на канве операторами, записанными в обработчике события **OnDrawSection** компонента **HeaderControl**.

Компонент **Header** обладает существенно меньшими возможностями, чем **HeaderControl**. В нем свойство **Sections** имеет тип **TStrings** и содержит только тексты заголовков, не позволяя регулировать пределы изменения ширины секций, их функционирование как кнопок и т.д. Таким образом, **Header** имеет смысл использовать только в 16-разрядных приложениях.

3.7.4 Многостраничные панели — компоненты **TabControl**, **PageControl**, **TabSet**, **TabbedNotebook**, **Notebook**

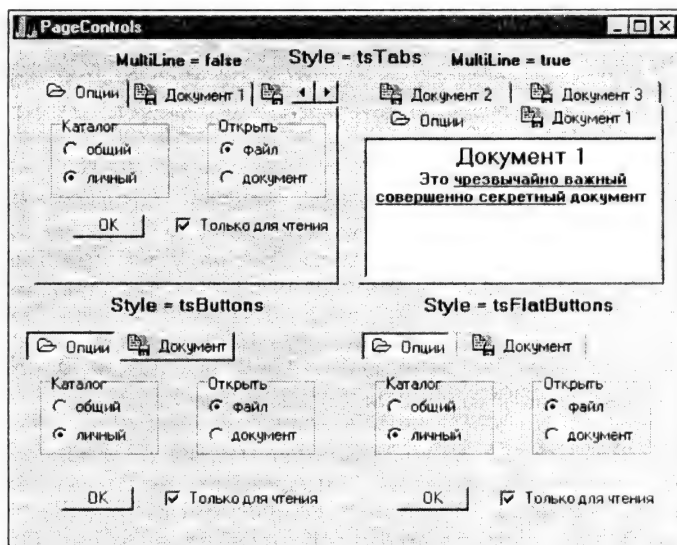
Многостраничные панели позволяют экономить пространство окна приложения, размещая на одном и том же месте страницы разного содержания. На рис. 3.40 показаны различные формы отображения многостраничного компонента **PageControl**. Начнем рассмотрение многостраничных панелей именно с этого компонента.

Перенесите компонент **PageControl** на форму. Чтобы задавать и редактировать страницы этого компонента, надо щелкнуть на нем правой кнопкой мыши. Во всплывшем меню вы можете видеть команды: **New Page** — создать новую страницу, **Next Page** — переключиться на следующую страницу, **Previous Page** — переключиться на предыдущую страницу.

Каждая создаваемая вами страница является объектом типа **TTabSheet**. Это панель, на которой можно размещать любые управляющие компоненты, окна редактирования и т.п. После того, как вы создадите несколько страниц, выделите одну из них, щелкнув в ее середине, и посмотрите ее свойства в Инспекторе Объектов. Страница имеет следующие основные свойства:

Рис. 3.40

Иллюстрация различных вариантов панели PageControl



Name	Имя, по которому можно ссылаться на страницу
Caption	Надпись, которая появляется на ярлычке закладки
PageIndex	Индекс страницы, по которому можно ссылаться на страницу
ImageIndex	Индекс изображения, которое может появляться на ярлычке закладки

Из общих свойств компонента **PageControl** можно отметить:

Style	Определяет стиль отображения компонента: tsTabs — закладки (верхние компоненты на рис. 3.40), tsButtons — кнопки (левый нижний компонент на рис. 3.40), tsFlatButtons — плоские кнопки (правый нижний компонент на рис. 3.40)
MultiLine	Определяет, будут ли закладки размещаться в несколько рядов, если все они не помещаются в один ряд (на рис. 3.40 сверху два одинаковых компонента, но в левом MultiLine = false , а в правом — true ; примером компонента с MultiLine = false является также знакомая вам палитра компонентов в C++Builder)
TabPosition	Определяет место расположения ярлычков закладок: tpBottom — внизу, tpLeft — слева, tpRight — справа и tpTop — вверху компонента (это значение по умолчанию и именно оно задано в примерах рис. 3.40)
TabHeight и TabWidth	Высота и ширина ярлычков закладок в пикселях. Если значения этих параметров заданы равными 0, то размеры ярлычков определяются автоматически по размерам надписей на них
Images	Ссылка на компонент ImageList (см. раздел 3.9.2), который содержит список изображений на ярлычках. Свойства ImageIndex страниц содержат индексы, соответствующие именно этому списку
ScrollOpposite	Определяет способ перемещения закладок при размещении их в несколько рядов (опробуйте экспериментально, как это свойство влияет на поведение компонента)

ActivePage	Имя активной страницы
Pages	Доступ к странице по индексу (первая страница имеет индекс 0).
[int Index]	Свойство только для чтения
PageCount	Количество страниц. Свойство только для чтения

В компоненте имеется ряд методов, позволяющих оперировать страницами, создавать их, уничтожать, переключать. Посмотрите их во встроенной справке C++Builder. Основные события компонента — **OnChanging** и **OnChange**. Первое из них происходит непосредственно перед переключением на другую страницу после щелчка пользователя на новой закладке. При этом в обработчик события передается по ссылке параметр **AllowChange** — разрешение переключения. Если в обработчике задать **AllowChange = false**, то переключение не произойдет. Событие **OnChange** происходит сразу после переключения.

Рассмотрим теперь компонент **TabControl**. Внешне этот компонент выглядит так же, как **PageControl**, и имеет много тех же свойств: **Style**, **MultiLine**, **TabPosition**, **TabHeight**, **TabWidth**, **Images**, **ScrollOpposite**, те же события **OnChanging** и **OnChange**. Но принципиальное отличие его от **PageControl** заключается в том, что **TabControl** не имеет множества панелей (страниц). Компонент представляет собой одну страницу с управляющим элементом типа кнопки со многими положениями. И надо написать соответствующие обработчики событий **OnChanging** и **OnChange**, чтобы определить, что именно должно происходить на панели при переключениях закладок пользователем. У компонента имеется еще одно свойство — **MultySelect**, позволяющее множественный выбор закладок. Если это свойство установлено в **true**, то в обработчиках событий надо описать реакцию на такой выбор пользователя.

Число закладок и их надписи определяются свойством **Tabs** типа **TStrings**. В нем вы можете задать надписи закладок. Сколько строчек надписей вы укажете, столько будет закладок. Текущее состояние переключателя определяется свойством **TabIndex**. Вы можете установить его в процессе проектирования, чтобы определить исходное состояние переключателя. А затем в обработчиках событий **OnChanging** и **OnChange** можете читать это свойство, чтобы определить, что именно выбрал пользователь.

Применять компонент **TabControl** имеет смысл в тех приложениях, в которых нужен многопозиционный переключатель. Вы можете, конечно, имитировать с помощью **TabControl** поведение, аналогичное компоненту **PageControl**. Для этого достаточно, например, расположить в пределах **TabControl** две закрывающие друг друга панели и в обработчик события **OnChange** вставить оператор:

```
if (TabControl1->TabIndex == 0)
    Panel2->Visible = false;
else Panel2->Visible = true;
```

Если **Panel2** — верхняя панель, то при выборе первой закладки (**TabIndex = 0**) она будет делаться невидимой и под ней будет проступать нижняя панель.

Но подобная имитация **PageControl** не имеет смысла, так как проще использовать сам компонент **PageControl**. А **TabControl** надо применять, если требуются какие-то перестроения в рамках одной панели.

Теперь коротко остановимся на компонентах **TabSet**, **TabbedNotebook** и **Notebook**. Эти компоненты не рекомендуются для применения в 32-разрядных приложениях.

Компонент **TabbedNotebook** является аналогом многостраничной панели **PageControl**. Только многие одинаковые у этих панелей свойства называются

по-разному. Основное свойство — **Pages**, определяющее число страниц и надписи закладок. Свойство **ActivePage** определяет надпись активной страницы. Свойство **PageIndex** определяет индекс активной страницы (0 — первая страница). Так что узнать, какая страница активна, можно или по значению **ActivePage**, или по **PageIndex**.

В обработчик события **OnChange**, происходящего при переключении пользователем страницы, передается параметр **NewTab**, равный индексу новой страницы, и **AllowChange** — разрешение переключения. Для запрета переключения можно в обработчике задать **AllowChange = false**.

Рассмотренный компонент **TabbedNotebook** является как бы соединением двух компонентов: пачки панелей (страниц) **Notebook** и набора закладок **TabSet**. Эти два компонента могут использоваться и отдельно. Компонент **TabSet** во многом аналогичен рассмотренному ранее 32-разрядному компоненту **TabControl**. Это многопозиционный управляющий элемент, который сам по себе не имеет никакой панели. Его основное свойство — **Tabs** типа **TStrings**. Задавая строки этого свойства вы тем самым определяете число закладок и их надписи. Свойства **StartMargin** и **EndMargin** определяют поля — расстояния крайних закладок от краев компонента. Сами закладки всегда направлены вниз. Поэтому компонент **TabSet** надо располагать внизу управляемого им компонента. Свойство **AutoScroll** определяет появление кнопок при большом количестве закладок, которые позволяют пользователю прокручивать полосу закладок, как это делается в компонентах **PageControl** и **TabControl** при **MultiLine = false**. Индекс выбранной закладки определяется свойством **TabIndex**, значение которого можно устанавливать и можно читать в обработчике события **OnChange**, происходящего при смене пользователем закладки и идентичного такому событию в компоненте **TabbedNotebook**.

Компонент **Notebook** является пачкой панелей, имена и количество которых определяются свойством **Pages**, как в компоненте **TabbedNotebook**. Индекс выбранной страницы определяется свойством **PageIndex**. В этом компоненте отсутствует управляющий элемент — закладки. Так что страницы можно переключать какими-то кнопками, переключать их в зависимости от действий пользователя, в зависимости от отображаемых данных и т.п. Компоненты **Notebook** и **TabSet** могут быть, конечно, объединены программно в компонент, аналогичный **TabbedNotebook**. Для этого достаточно в обработчик события **OnChange** компонента **TabSet** вставить оператор

```
Notebook1->PageIndex = NewTab;
```

Но подобное использование этих компонентов вряд ли целесообразно: уж лучше использовать непосредственно **TabbedNotebook**.

3.7.5 Инструментальные панели — компоненты **ToolBar** и **PageScroller**

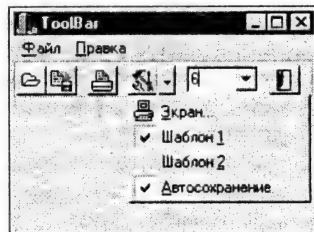
Как уже говорилось выше, инструментальные панели можно создавать, не прибегая к специальным компонентам. Можно поместить на форму простейшую панель **Panel**, разместить на ней быстрые кнопки **SpeedButton** и панель готова. Остается только написать для кнопок соответствующий код. Но специализированные компоненты, которые мы рассмотрим в этом разделе, дают, конечно, дополнительные возможности для построения инструментальных панелей.

Начнем рассмотрение компонентов, которые используются для построения различных инструментальных панелей, с компонента **ToolBar**. Пример панели, построенной на основе этого компонента, приведен на рис. 3.41.

Если вы поместите компонент **ToolBar** на форму, то по умолчанию он расположится сверху, поскольку его свойство **Align** по умолчанию равно **alTop**. Если вы

Рис. 3.41

Пример инструментальной панели **ToolBar**



хотите, чтобы панель располагалась иначе, установите **Align = alNone**, после чего можете придать панели любую форму и расположить ее в любом месте.

Занесение компонентов на панель **ToolBar** можно, в принципе, осуществлять обычным способом — переносом их из палитры компонентов. Но для занесения кнопок имеется и более простой вариант. Щелкните на **ToolBar** правой кнопкой мыши и выберите из всплывшего меню команду **New Button**. На форме появится очередная кнопка — объект типа **TToolButton**. Это не совсем обычная кнопка, так как в дальнейшем вы увидите, что внешне она может не походить на кнопку. Ее вид и поведение определяется ее свойством **Style**, которое по умолчанию равно **tbsButton** — кнопка. Другие возможные стили мы рассмотрим позднее. А как кнопка этот объект очень похож на кнопку **SpeedButton**. Только изображение на кнопке определяется не свойством **Glyph**, а свойством **ImageIndex**. Оно определяет индекс изображения, хранящегося во внешнем компоненте **ImageList** (см. раздел 3.9.2). Указание на этот компонент может задаваться такими свойствами компонента **ToolBar**, как **Images**, **DisabledImages** (указывает на список изображений кнопок в недоступном состоянии) и **HotImages** (указывает на список изображений кнопок в моменты, когда над ними перемещается курсор мыши).

Свойство **MenuItem** позволяет задать раздел главного или контекстного меню (см. разделы 3.6.1 и 3.6.2), который дублируется данной кнопкой. При установке этого свойства, если в соответствующем разделе меню было задано изображение и установлен текст подсказок (свойство **Hint**), то это же изображение появится на кнопке и тот же текст появится в свойстве **Hint** кнопки. Передадутся из раздела меню в кнопку также значения свойств **Enabled** (доступность) и **Visible** (видимость). Правда, все это передастся в кнопку только в момент установки свойства **MenuItem**. Если в процессе дальнейшего проектирования вы измените соответствующие свойства раздела меню, это не отразится на свойствах кнопки. Но если вы сотрете значение **MenuItem**, а потом установите его снова, то в кнопке зафиксируются новые значения свойств раздела меню.

Свойство **Wrap**, установленное в **true**, приводит к тому, что после этой кнопки ряд кнопок на панели прерывается и следующие кнопки размещаются в следующем ряду.

Теперь вернемся к свойству **Style**, задающему стиль кнопки. Значение **Style = tbsCheck** определяет, что после щелчка пользователя на кнопке она остается в нажатом состоянии. Повторный щелчок на кнопке возвращает ее в отжатое состояние. Поведение такой кнопки подобно кнопкам **SpeedButton** и определяется аналогичными свойствами **AllowAllUp** и **Down** (см. раздел 3.5.2). Если при этом в нескольких кнопках установлено свойство **Grouped = true**, то эти кнопки образуют группу, из которой только одна кнопка может находиться в нажатом состоянии.

Значение **Style = tbsDropDown** соответствует кнопке в виде выпадающего списка. Этот стиль удобен для воспроизведения выпадающего меню. Если для подобной кнопки задать в качестве свойства **MenuItem** головной раздел меню, то в выпадающем списке автоматически будут появляться разделы выпадающего меню. В примере рис. 3.41 стиль **tbsDropDown** имеет четвертая слева кнопка. В ней в качестве свойства **MenuItem** задан раздел **Опции** из меню, рассмотренного в разделе 3.6.1 и представленного на рис. 3.33. При **Style = tbsDropDown** можно вместо

свойства **MenuItem** задать свойство **DropDownMenu**, определяющее контекстное меню (компонент **PopupMenu**), которое будет отображаться в выпадающем списке.

Значение **Style = tbsSeparator** приводит к появлению разделителя, позволяющего отделить друг от друга кнопки разных функциональных групп. Значение **Style = tbsDivider** приводит к появлению разделителя другого типа — в виде вертикальной линии. Разделитель можно ввести из контекстного меню **ToolBar**, выбрав команду **New Separator**.

Свойство кнопки **Indeterminate** задает ее третье состояние — не нажатая и не отпущенная. Это свойство можно устанавливать в **true** во время выполнения, если в данном режиме кнопка не доступна.

Свойство **Marked** выделяет кнопку.

Мы рассмотрели занесение на панель кнопок. Но в инструментальных панелях нередко используются и выпадающие списки. Например, для задания размера шрифта. Не представляет труда перенести на панель **ToolBar** такие компоненты, как **ComboBox** (это изображено на рис. 3.41), **SpinEdit** и др.

Из общих свойств компонента **ToolBar** следует еще отметить **ButtonHeight** и **ButtonWidth** — высота и ширина кнопок в пикселях, и **Wrapable** — автоматический перенос кнопок в следующий ряд панели, если они не помещаются в предыдущем. Такой перенос осуществляется и во время проектирования, и во время выполнения при изменении пользователем размеров панели.

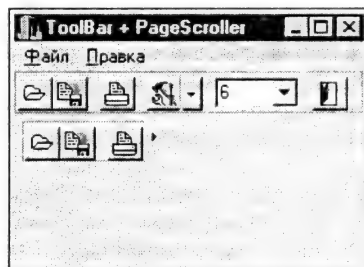
Свойства, определяющие вид панели **ToolBar**: **BorderWidth** — ширина бордюра, **EdgeInner** и **EdgeOuter** — стиль изображения внутренней и внешней части панели (утопленный или выступающий), **EdgeBorders** — определяет изображение отдельных сторон панели (левой, правой, верхней, нижней).

Мы рассмотрели построение инструментальной панели на основе компонента **ToolBar**. Но полоса может быть очень длинной и не помещаться в отведенном ей месте формы. Примером является палитра компонентов C++Builder. В этих случаях может помочь компонент **PageScroller**, обеспечивающий прокрутку панели. Собственно говоря, **PageScroller** может прокручивать любой компонент, не обязательно панель **ToolBar**. Например, он может прокручивать какую-то панель вместе с размещенными на ней компонентами. В этом отношении он напоминает рассмотренный в разделе 3.7.2 компонент **ScrollBar**. Но есть и заметные различия между этими двумя компонентами: **PageScroller** прокручивает только один компонент и только в одном направлении — горизонтальном или вертикальном. Да и оформление управления прокруткой у **PageScroller** не похоже на полосы прокрутки в **ScrollBar**.

Пример применения компонента **PageScroller** показан на рис. 3.42. Это тот же пример, что и на рис. 3.41. В верхней части окна показана та же инструментальная панель, что и на рис. 3.41. А ниже показана идентичная панель, но заключенная в небольшое окно **PageScroller** и снабженная кнопкой прокрутки.

Рис. 3.42

Пример инструментальной панели и ее прокрутки компонентом **PageScroller**



Основное свойство компонента **PageScroller** — **Control**. Оно указывает компонент, который должен размещаться и прокручиваться в окне **PageScroller**. Благо-

даря наличие этого свойства вы можете проектировать свою инструментальную панель, например, **ToolBar**, не помещая ее заранее в окно **PageScroller** и не задумываясь о ее размере. А после того, как вы спроектировали панель, можно ввести на форму компонент **PageScroller** и установить его свойство **Control**. В этот момент ваша инструментальная панель переместится в окно компонента **PageScroller** и появится, если необходимо, кнопка прокрутки.

Свойство **Margin** компонента **PageScroller** определяет размер полей в пикселях, которые оставляются между краем окна **PageScroller** и прокручиваемым компонентом. По умолчанию эти поля равны нулю, но надо задать свойству **Margin** некоторое положительное значение. Иначе края прокручиваемого компонента могут быть плохо видны.

Свойство **AutoScroll** определяет, должна ли прокрутка осуществляться автоматически, как только курсор мыши пройдет над кнопкой прокрутки. Опробуйте режим автоматической прокрутки экспериментально. На мой взгляд лучше оставлять значение **AutoScroll** равным **false**, поскольку такая автоматическая прокрутка не очень удобна пользователю.

3.7.6 Перестраиваемые панели — компоненты **CoolBar** и **ControlBar**

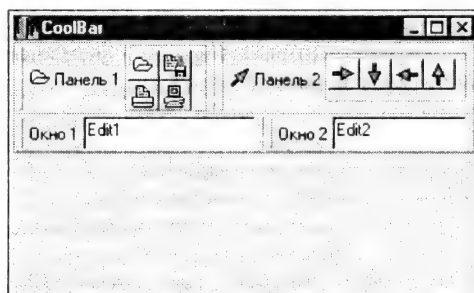
Перестраиваемые панели являются дальнейшим развитием инструментальных панелей. Только в перестраиваемых панелях сами инструментальные панели обычно являются компонентами более сложных образований. Примером перестраиваемой панели может служить панель ИСП C++Builder 5, включающая в себя ряд более мелких панелей быстрых кнопок и палитру компонентов. Пользователь может настраивать их, изменять местоположение панелей и т.п.

Начнем рассмотрение с компонента **CoolBar**. Он позволяет строить перестраиваемые панели, состоящие из полос (bands). В полосы могут включаться инструментальные панели **ToolBar** и любые другие оконные компоненты: окна редактирования, панели и т.п. Каждый из этих компонентов автоматически снабжается средствами перемещения его пользователем в пределах окна **CoolBar**. В полосы могут вставляться и не оконные компоненты, например, метки. Но они не будут перемещаемыми.

Опробуйте в работе этот компонент. Поместите его на форму. Перенесите на него другие компоненты, например, **ToolBar** и **Edit**. Когда вы размещаете на **CoolBar** очередной компонент, ему отводится отдельная полоса и он растягивается на всю ширину **CoolBar**. Около каждого компонента появляется слева полоска, за которую компонент можно перемещать. Например, взявшись за эту полоску вы можете переместить полосу вместе с ее компонентом в тот ряд, где уже имеется другой компонент. Тогда они расположатся в ряд один левее другого (см. пример на рис. 3.43).

Рис. 3.43

Пример перестраиваемой панели на основе компонента **CoolBar**



Свойства полос вы можете задавать редактором полос. С этим инструментом вы уже имели дело в разделах 3.4.2 и 3.7.3 (см. рис. 3.17). Вызвать редактор полос можно тремя способами: из Инспектора Объектов кнопкой с многоточием около свойства **Bands**, двойным щелчком на компоненте **CoolBar** или из контекстного меню, выбрав команду **Bands Editor**. В окне этого редактора вы можете перемещаться по полосам, добавлять новые полосы или уничтожать существующие. При перемещении по полосам в окне Инспектора Объектов вы будете видеть свойства полос. Свойство **Control** определяет размещенный на полосе компонент. Свойство **Break** определяет, занимает ли полоса весь соответствующий размер контейнера **CoolBar**, или обрывается. Если вы расположите полосы так, как показано на рис. 3.43, то в левых полосах автоматически установится **Break = true**, а в правых — **Break = false**.

Свойство **Text** задает текст, который может появиться в начале соответствующей полосы. Это свойство можно оставлять пустым. А можно и задать его — надписи «Панель 1», «Панель 2», «Окно 1», «Окно 2» на рис. 3.43 заданы именно этим свойством.

Вместо свойства **Text** (или наряду с ним) можно задать свойство **ImageIndex** — индекс списка изображений **ImageList** (см. раздел 3.9.2), ссылка на который задается свойством **Images**. Указанные таким образом изображения появятся в начале соответствующих полос (см. верхние полосы на рис. 3.43).

Свойства **MinHeight** и **MinWidth** определяют минимальную высоту и ширину полосы при перестроениях пользователем полос панели.

Свойство **FixedSize** определяет, фиксирован ли размер данной полосы или он может изменяться пользователем. По умолчанию для всех полос **FixedSize = false**, т.е. все полосы перестраиваются. Но при желании размеры некоторых полос можно зафиксировать, задав для них **FixedSize = true**.

Для компонента **CoolBar** в целом, помимо обычных для других панелей свойств, надо обратить внимание на свойство **BandMaximize**. Оно определяет действие, которым пользователь может установить максимальный размер полосы, не перетаскивая ее границу: **bmNone** — такое действие не предусмотрено, **bmClick** — щелчком мыши, **bmDbClick** — двойным щелчком. Наиболее целесообразно, по-видимому, задавать значения **bmDbClick** или **bmNone**, поскольку значение **bmClick** приводит к резкому перестроению полос даже при случайном щелчке мыши.

Свойство **FixedOrder**, если его установить в **true**, не разрешит пользователю в процессе перемещений полос изменять их последовательность. Вероятно, такое задание лучше, чем значение по умолчанию, равное **false**, поскольку чрезмерная свобода для пользователя способна его запутать.

Свойство **Vertical** указывает вертикальное или горизонтальное расположение полос. По умолчанию **Vertical = false**, что соответствует горизонтальным полосам.

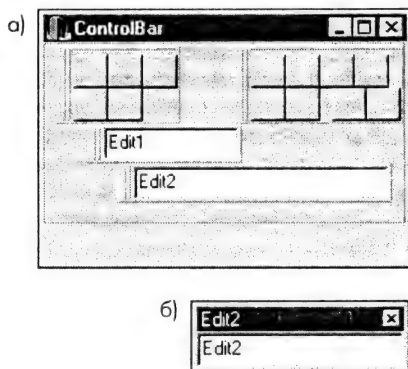
Запустите свое тестовое приложение, если вы его построили, и опробуйте его в работе. Вы увидите, как легко пользователь может перестраивать панель.

Еще большую свободу перестроений дает пользователю компонент **ControlBar**. Только оформление панели несколько отличается от **CoolBar** и кроме того в ней может широко применяться техника перетаскивания и встраивания **Drag&Doc**, подробно рассмотренная в главе 4 в разделе 4.4.2. Поэтому ограничимся пока кратким описанием панели **ControlBar**.

Поместите на форму компонент **ControlBar** и перенесите на него несколько компонентов, например, инструментальных панелей **ToolBar** и окон редактирования **Edit**. Вы увидите (см. рис. 3.44), что каждый компонент, попадая на **ControlBar**, получает полосу захвата, свойственную технологии **Drag&Doc**.

Рис. 3.44

Перестраиваемая панель на основе компонента **ControlBar** (а) и вынесенное из нее окно редактирования (б)



Установите у компонентов, размещенных на **ControlBar**, свойство **DragMode** = **dmAutomatic** и **DragKind** = **dkDock**. Это означает автоматическое выполнение операций **Drag&Doc** (см. раздел 4.4.2).

Свойства компонента **ControlBar** **RowSize** и **RowSnap** определяют процедуру встраивания. Свойство **RowSize** задает размеры полос, в которые могут встраиваться компоненты, а **RowSnap** определяет захват полосами встраиваемых компонентов. Свойство **AutoDrag** определяет, можно (при значении **true**), или нельзя простым перетаскиванием вынести полосу за пределы **ControlBar**.

Запустите приложение и посмотрите на практике широчайшие возможности перестроения панелей. А если вы установили свойство **AutoDrag** в **true**, то вы можете даже вынимать из панели отдельные компоненты и они становятся самостоятельными окнами (см. рис. 3.44 б). Опробуйте в эксперименте различные значения свойств компонента **ControlBar**, и они станут вам более понятны. А подробнее о технике **Drag&Doc** вы узнаете в разделе 4.4.2.

3.7.7 Полоса состояния **StatusBar**

Компонент **StatusBar** представляет собой ряд панелей, отображающих полосу состояния в стиле Windows. Обычно эта полоса размещается внизу формы. Пример полосы состояния вы можете увидеть в разделе 3.2.4 на рис. 3.5.

Свойство **SimplePanel** определяет, включает ли полоса состояния одну или множество панелей. Если **SimplePanel** = **true**, то вся полоса состояния представляет собой единственную панель, текст которой задается свойством **SimpleText**. Если же **SimplePanel** = **false**, то полоса состояния является набором панелей, задаваемых свойством **Panels**. В этом случае свойство **SizeGrip** определяет, может ли пользователь изменять размеры панелей в процессе выполнения приложения.

Каждая панель полосы состояния является объектом типа **TStatusPanels**. Свойства панелей вы можете задавать специальным редактором наборов. С этим инструментом вы уже имели дело в разделах 3.4.2, 3.7.3, 3.7.6 (см. рис. 3.17). Вызвать редактор можно тремя способами: из Инспектора Объектов кнопкой с многоточием около свойства **Panels**, двойным щелчком на компоненте **StatusBar** или из контекстного меню, выбрав команду **Panels Editor**. В окне редактора вы можете перемещаться по панелям, добавлять новые или уничтожать существующие. При перемещении по панелям в окне Инспектора Объектов вы будете видеть их свойства.

Основное свойство каждой панели — **Text**, в который заносится отображаемый в панели текст. Его можно занести в процессе проектирования, а затем можно изменять программно во время выполнения. Другое существенное свойство панели — **Width** (ширина).

Программный доступ к текстам отдельных панелей можно осуществлять через свойство **Panels** и его индексированное подсвойство **Items**. Например, оператор:


```
StatusBar1->Panels->Items[0]->Text = "текст 1";
```

напечатает текст «текст 1» в первой панели.

Количество панелей полосы состояния можно определить из подсвойства **Count** свойства **Panels**. Например, следующий оператор очищает тексты всех панелей:

```
for (int i = 0; i < StatusBar1->Panels->Count; i++)
{
    StatusBar1->Panels->Items[i]->Text = "";
}
```

На рис. 3.5 был приведен пример текстового редактора на основе компонента **RichEdit**, содержащий полосу состояния. В ее первой панели отображается номер строки и символа, перед которым находится курсор, во второй — отображается, модифицирован текст в окне, или нет. В третьей панели отображается подсказка о назначении компонента, над которым в данный момент расположен курсор мыши.

Для реализации такой полосы состояния надо в обработчиках событий **OnKeyDown**, **OnKeyUp**, **OnMouseDown** и **OnMouseUp** компонента **RichEdit1** и события **OnResize** формы обеспечить выполнение операторов:

```
StatusBar1->Panels->Items[0]->Text =
    IntToStr((int)RichEdit1->CaretPos.y+1) +
    ": "+IntToStr((int)RichEdit1->CaretPos.x+1);
if (RichEdit1->Modified)
    StatusBar1->Panels->Items[1]->Text = "модиф.";
else StatusBar1->Panels->Items[1]->Text = "";
```

Эти операторы заполняют первые две панели полосы состояния. Занесение в строку состояния подсказок описано в разделе 4.1.9.

3.7.8 Фреймы

В C++Builder 5 введен новый компонент, который помогает поддерживать стилистическое единство приложения. Это **Frame** — фрейм. Он представляет собой нечто среднее между панелью и формой. С формой его роднит то, что он:

- проектируется отдельно, как самостоятельное окно
- имеет свой модуль — файл **.cpp**
- имеет возможности наследования, причем даже более широкие, чем у формы, так как может наследоваться даже внутри одного приложения
- может включаться в Депозитарий и использоваться так же, как и форма, включая наследование

С панелью фрейм роднит то, что он:

- не является самостоятельным окном Windows и может отображаться только на форме или другом контейнере
- имеет свойства, методы, события, подобные панели, а не форме

Таким образом, фрейм — это панель, т.е. некий фрагмент окна приложения, но способный переноситься на разные формы, в разные приложения и допускающий использование преимуществ наследования.

Начать проектирование нового фрейма можно командой **File | New Frame** или командой **File | New** и выбором пиктограммы **Frame** на странице **New** окна **Депозитария**. В обоих случаях перед вами откроется окно фрейма, подобное окну формы, а в Редакторе Кода вы увидите текст заготовки модуля фрейма:

```
...
#include "Unit2.h"
...
TFrame2 *Frame2;
```



```
/*
Сюда могут помещаться объявления типов, констант, переменных, к которым
не будет доступа из других модулей. Они будут едины для всех объектов
фреймов. Тут же должны быть реализации всех объявленных в заголовочном
файле функций, а также могут быть реализации любых дополнительных, не
объявленных ранее функций
*/
```

```
*/
```

```
// Заготовка конструктора
__fastcall TFrame2::TFrame2(TComponent* Owner)
: TFrame(Owner)
{
}
```

Если, щелкнув правой кнопкой мыши в окне Редактора Кода вы выберете в контекстном меню раздел Open Source\Header File, то увидите заготовку заголовочного файла модуля фрейма:

```
...
// Объявление класса фрейма
class TFrame2 : public TFrame
{
    _published:      // IDE-managed Components
/*
Сюда C++Builder помещает объявления компонентов, размещаемых на фрейме.
Не добавляйте сюда ничего вручную
*/
private: // User declarations
/*
Закрытый раздел класса. Сюда могут помещаться объявления переменных и
функций, включаемых в класс фрейма, но не доступных для других модулей
*/
public:      // User declarations
/*
Открытый раздел класса. Сюда могут помещаться объявления переменных и
функций, включаемых в класс фрейма и доступных для других модулей
*/
// Объявление конструктора
    __fastcall TFrame2(TComponent* Owner);
};
...
```

Комментарии в приведенном тексте поясняют, куда и что можно помещать в модуле. Те переменные, объявления которых вы поместите в объявление класса, будут индивидуальны для каждого объекта фрейма. Объявления имеют обычный для класса вид. Например:

```
int A;
```

Если вы хотите ввести переменную, общую для всех объектов фреймов, ее надо объявить со спецификатором **static**. Например:

```
static int A;
```

В этом случае надо не забыть инициализировать эту переменную вне объявления класса, например, оператором

```
int TFrame2::A = 0;
```

Здесь для доступа к статической переменной использовано имя класса **TFrame2** и бинарная операция разрешения области действия «::». Если не сделать такой инициализации, то будет выдано сообщение компилятора о неразрешенной внешней ссылке и программа не будет скомпилирована. Те, кому не очень понятны эти операции со статическими элементами-данными класса, могут найти подробные разъяснения всех вопросов, связанных с объявлением классов, в главе 13.

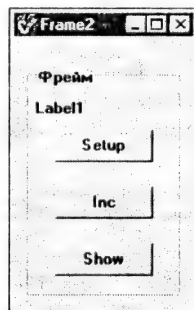
На фрейм вы можете так же, как на форму, переносить и размещать любые компоненты, устанавливать их свойства, писать обработчики их событий и т.п.

Давайте создадим чисто тестовый фрейм, чтобы на его примере продемонстрировать проектирование фрейма, его использование, доступ к различным его элементам и наследование свойств.

Начните новое приложение и выполните команду **File | New Frame**. Перенесите на фрейм групповую панель **GroupBox** (см. раздел 3.5.3). Перенесите на панель метку **Label** и три кнопки **Button**. Разместите все эти компоненты примерно так, как показано на рис. 3.45, изменив соответственно их надписи (**Caption**) и назвав кнопки соответственно **BSetup**, **BInc**, **BShow**.

Рис. 3.45

Пример фрейма



Давайте введем в наш модуль в разных местах объявления целых переменных, а в обработчики событий кнопок введем операторы, манипулирующие ими и отображающие результат в метке. Заголовочный файл фрейма должен приобрести следующий вид:

```
...
class TFrame2 : public TFrame
{
    __published:          // IDE-managed Components
        TGroupBox *GroupBox1;
        TLabel *Label1;
        TButton *BSetup;
        TButton *BInc;
        TButton *BShow;
        void __fastcall BSetupClick(TObject *Sender);
        void __fastcall BIncClick(TObject *Sender);
        void __fastcall BShowClick(TObject *Sender);
private:                // User declarations
    //Переменная A видна только в данном модуле
    int A;
public:                  // User declarations
    //Переменные B и C видны в других модулях через объект фрейма
    int B;
    static int C;
    __fastcall TFrame2(TComponent* Owner);
};
//Инициализация статической переменной
int TFrame2::C = 1;
...
```

Файл реализации модуля фрейма должен иметь вид:

```
...
TFrame2 *Frame2;
//Переменная D видна только в данном модуле
int D;
//_____
```

```

__fastcall TFrame2::TFrame2(TComponent* Owner)
    : TFrame(Owner)
{
}
//-----

void __fastcall TFrame2::BSetupClick(TObject *Sender)
{
    A = 1;
    B = 1;
    C = 1;
    D = 1;
    Label1->Caption = "A="+IntToStr(A)+ " B="+IntToStr(B)+
        " C="+IntToStr(C)+ " D="+IntToStr(D);
}
//-----

void __fastcall TFrame2::BIncClick(TObject *Sender)
{
    A += 1;
    B += 1;
    C += 1;
    D += 1;
    Label1->Caption = "A="+IntToStr(A)+ " B="+IntToStr(B)+
        " C="+IntToStr(C)+ " D="+IntToStr(D);
}
//-----

void __fastcall TFrame2::BShowClick(TObject *Sender)
{
    Label1->Caption = "A="+IntToStr(A)+ " B="+IntToStr(B)+
        " C="+IntToStr(C)+ " D="+IntToStr(D);
}

```

В модуле введены переменные:

- **A** — введена в закрытый раздел класса; видна только в процедурах данного класса в этом модуле; независимые друг от друга переменные **A** будут содержаться в каждом объекте фрейма
- **B** — введена в открытый раздел класса; в других модулях можно получить доступ к **B** через имя объекта фрейма; независимые друг от друга переменные **B** будут содержаться в каждом объекте фрейма
- **C** — введена в открытый раздел класса как статическая переменная; в других модулях можно получить доступ к **C** через имя класса фрейма с помощью бинарной операции разрешения области действия «::» или через имя любого объекта фрейма; имеется единственный экземпляр **C**, независимо от числа объектов фреймов
- **D** — введена в реализацию класса; доступна только в данном модуле; имеется единственный экземпляр **D**, независимо от числа объектов фреймов

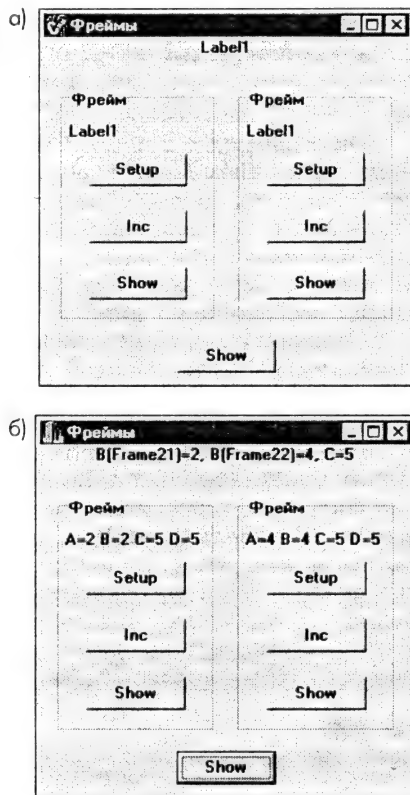
Введенные в модуль обработчики щелчков на кнопках обеспечивают сброс всех переменных в 1 (процедура **TFrame2::BSetupClick**), увеличение всех переменных на 1 (процедура **TFrame2::BIncClick**), отображение текущего состояния переменных (процедура **TFrame2::BShowClick**).

Теперь давайте разместим несколько экземпляров фрейма на форме. Перейдите в основную форму приложения и выберите в палитре компонентов **Frame** (первая кнопка на странице *Standard*). Появится диалоговое окно, в котором будет спрашиваться, какой фрейм вы хотите разместить на форме. Выберите ваш фрейм **Frame2** и он появится на форме. Можете отбуксировать его, как обычный компонент, в нужное место. Повторите эту операцию еще раз и разместите на форме вто-

рой фрейм (рис. 3.46). Добавьте внизу формы кнопку Show, а сверху — метку, задав ее свойство **Align** равным **alTop** и свойство **Alignment** равным **taCenter**.

Рис. 3.46

Пример использования фреймов: форма (а) и приложение в работе (б)



Вы получили форму, содержащую два объекта — фрейма. Можете изменить какие-то свойства объектов. Например, изменить надписи (**Caption**) групповых панелей **GroupBox** (см. рис. 3.46 а). После того, как вы изменили эти свойства, они перестают наследоваться из класса фрейма. А остальные свойства продолжают наследоваться. В этом легко убедиться. Перейдите в модуль фрейма (рис. 3.45) и измените у фрейма стиль шрифта (**Font->Style**) на жирный. Вы увидите, что в обоих объектах главной формы шрифт тоже станет жирным. Верните во фрейме шрифт на обычный и он синхронно изменится в объектах. А теперь установите в одном из фреймов на форме шрифт жирным. Повторив после этого эксперимент с изменением шрифта в исходном фрейме, вы увидите, что теперь шрифт меняется только в том объекте формы, в котором вы его не изменяли вручную. Таким образом объекты наследуют только те свойства, которые не были в них установлены вручную.

Теперь давайте напишем обработчик щелчка на кнопке главной формы. Прежде всего взгляните на текст заголовочного файла модуля вышей формы (щелкните правой кнопкой мыши в окне Редактора Кода и выберите в контекстном меню раздел **Open Source\Header File**). Вы увидите, что в нем в описании класса формы появились две строки:

```
TFrame2 *Frame21;
TFrame2 *Frame22;
```

Это объявления указателей на объекты фреймов. Все компоненты, размещенные на фреймах, напрямую из модуля формы не видны. Доступ к ним можно полу-

чить только через объекты **Frame21** и **Frame22**. Имена компонентов, размещенных во фреймах, локальные. Несмотря на то, что во фреймах имеются кнопки с именами **BShow**, вы можете назвать тем же именем кнопку на форме.

Поместите в обработчик щелчка на этой кнопке оператор

```
Label1->Caption = "B(Frame21)="+IntToStr(Frame21->B)+
    ", B(Frame22)="+IntToStr(Frame22->B)+
    ", C="+IntToStr(TFrame2::C);
```

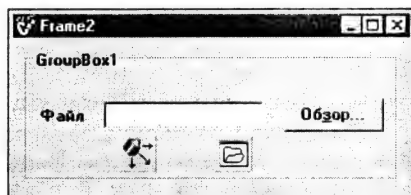
Он отображает в метке **Label1** значения переменных **B** объектов фреймов. Обращение к переменной **C** осуществлено через имя класса фрейма **TFrame2::C**. К ней можно было бы обратиться и через объекты фреймов: **Frame21->C** или **Frame22->C**. Все три формы обращения дают один и тот же результат. Значения переменных **A** и **D** отобразить невозможно, поскольку эти переменные недоступны из внешних модулей. Если вы попытаетесь отобразить их значения, компилятор выдаст сообщение об ошибке.

Сохраните ваше приложение, оттранслируйте его и выполните. Манипулируя кнопками, вы легко сможете убедиться (см. рис. 3.46 б), что переменные **A** и **B** независимы для каждого фрейма, а переменные **C** и **D** одинаковы. Точнее оба фрейма оперируют с одними и теми же переменными **C** и **D**.

Рассмотренный фрейм не имел никакого практического значения. Давайте построим более полезный пример. Во многих диалогах при установке различных опций фигурирует фрагмент, фрейм которого показан на рис. 3.47. Фрагмент включает в себя панель **GroupBox**, окно редактирования, в котором пользователь может написать имя файла, и кнопку **Обзор**, которая позволяет выбрать файл в стандартном диалоге **Windows** открытия файла. Если путь к файлу длинный, то полное имя файла с путем может не помещаться в окне редактирования. Поэтому полезно для него предусмотреть всплывающее окно, которое отображало бы полное имя файла вместе с путем и всплывало бы, если пользователь задержал над ним курсор мыши.

Рис. 3.47

Фрейм выбора файла



Давайте построим подобный фрейм и опробуем его в работе. Начните новое приложение и выполните команду **File | New Frame**. Перенесите на фрейм групповую панель **GroupBox**. Перенесите в эту панель окно редактирования **Edit**, кнопку **Button**, метку, диалог **OpenDialog** (см. раздел 3.8.2) и компонент **ApplicationEvents** — перехватчик событий приложения (см. раздел 3.9.3). Расположите компоненты примерно так, как показано на рис. 3.47.

Задайте в свойстве **Filter** диалога **OpenDialog** какой-то фильтр файлов, например, «все файлы|*.*.». Свойство **ShowHint** (показать ярлычок подсказки) в компонентах **Edit** и **Button** установите в **true**. В кнопке **Button** кроме того можете написать текст подсказки **Hint**, например, «Выбор файла|Выбор файла из каталога».

В обработчик события **OnShowHint** компонента **ApplicationEvents** занесите оператор:

```
if (HintInfo.HintControl == Edit1)
    if (Canvas->TextWidth(Edit1->Text) > Edit1->ClientWidth)
    {
        HintStr = Edit1->Text;
        ApplicationEvents1->CancelDispatch();
    }
```

Этот оператор в момент, когда должен отображаться ярлычок, проверяет, не является ли источником этого события (**HintInfo.HintControl**) окно редактирования **Edit1**. Если да, то проверяется, не превышает ли длина текста длины клиентской области **Edit1**. Если превышает, то текст ярлычка (**HintStr**) подменяется текстом, содержащимся в окне редактирования и принимаются меры (метод **CancelDispatch**), чтобы это событие не обрабатывалось другими компонентами **ApplicationEvents**, которые могут присутствовать в приложении. Пояснение всех этих операций см. в разделе 3.9.3.

Теперь введите в модуль фрейма глобальную переменную **FileName** типа **string**, в которой будет храниться имя выбранного файла. В обработчик щелчка на кнопке введите оператор

```
if (OpenDialog1->Execute())
{
    Edit1->Text = OpenDialog1->FileName;
    FileName = OpenDialog1->FileName;
}
```

который вызывает диалог открытия файла и помещает в окно редактирования **Edit1** и в переменную **FileName** имя файла, выбранного пользователем, вместе с путем к нему.

В обработчик события **OnExit** компонента **Edit1** поместите оператор

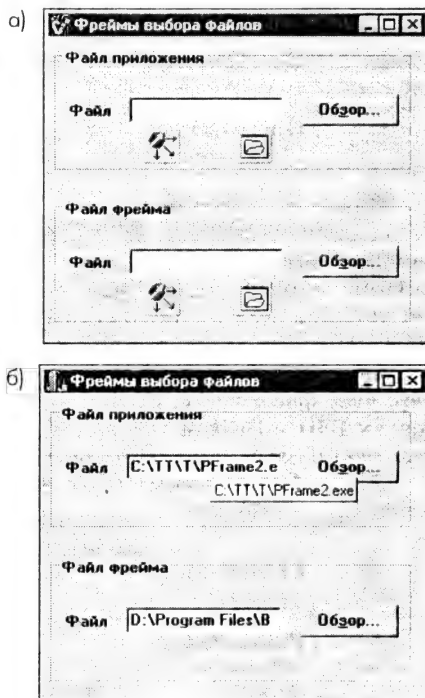
```
FileName = Edit1->Text;
```

заносящий в переменную **FileName** имя файла, если пользователь не пользовался диалогом, а просто написал в окне имя файла.

Программирование фрейма закончено. Теперь создайте тестовую программу, использующую этот фрейм. Предположим, что вам нужно разместить на форме два фрагмента, описанных вами во фрейме. Перейдите в основной модуль вашего приложения и разместите на форме так, как вы уже делали, два объекта вашего фрейма (рис. 3.48 а).

Рис. 3.48

Приложение с двумя фреймами выбора файла: его форма (а) и приложение во время выполнения (б)



Теперь вы можете поменять что-то в размещенных на форме объектах фреймов, изменить надписи групповых панелей, шрифты и т.п. Сохраните ваше приложение вместе с модулем фрейма, оттранслируйте его и проверьте в работе (рис. 3.48 б).

Вы разработали достаточно полезный фрейм и можете сохранить его в Депозитарии, чтобы в дальнейшем использовать в различных приложениях. Причем, если вы что-то измените во фрейме, хранящемся в Депозитарии, то все приложения, в которых использован этот фрейм, будут наследовать эти изменения. Благодаря использованию наследования, фрейм позволяет обеспечить единство стилистических решений не только внутри приложения, но и в рамках серии разрабатываемых вами приложений. Вам достаточно один раз разработать какие-то часто применяемые фреймы, включить их в Депозитарий, а затем вы можете использовать их многократно во всех своих проектах. Подробности о внесении в Депозитарий, о заимствовании из него форм и фреймов и о наследовании их свойств см. в главе 7 в разделе 7.4.








3.8 Системные диалоги

3.8.1 Общая характеристика компонентов — диалогов

В приложениях часто приходится выполнять стандартные действия: открывать и сохранять файлы, задавать атрибуты шрифтов, выбирать цвета палитры, производить контекстный поиск и замену и т.п.

Разработчики C++Builder позаботились о том, чтобы включить в библиотеку простые для использования компоненты, реализующие соответствующие диалоговые окна. Они размещены на странице Dialogs. В таблице 3.6 приведен перечень этих диалогов.

Таблица 3.6. Системные диалоги и их фрагменты

Пиктограмма	Компонент	Страница	Описание
	OpenDialog «Открыть файл»	Dialogs	Предназначен для создания окна диалога «Открыть файл».
	SaveDialog «Сохранить файл»	Dialogs	Предназначен для создания окна диалога «Сохранить файл».
	OpenPictureDialog «Открыть рисунок»	Dialogs	Предназначен для создания окна диалога «Открыть рисунок», открывающего графический файл.
	SavePictureDialog «Сохранить рисунок»	Dialogs	Предназначен для создания окна диалога «Сохранить рисунок» — сохранение изображения в графическом файле.
	FontDialog «Шрифты»	Dialogs	Предназначен для создания окна диалога «Шрифты» — выбор атрибутов шрифта.
	ColorDialog «Цвет»	Dialogs	Предназначен для создания окна диалога «Цвет» — выбор цвета.
	PrintDialog «Печать»	Dialogs	Предназначен для создания окна диалога «Печать».

Пиктограмма	Компонент	Страница	Описание
	PrinterSetupDialog «Установка принтера»	Dialogs	Предназначен для создания окна диалога «Установка принтера».
	FindDialog «Найти»	Dialogs	Предназначен для создания окна диалога «Найти» — контекстный поиск в тексте.
	ReplaceDialog «Заменить»	Dialogs	Предназначен для создания окна диалога «Заменить» — контекстная замена фрагментов текста.
	FileListBox (список файлов)	Win 3.1	Отображает список всех файлов каталога.
	DirectoryListBox (структура каталогов)	Win 3.1	Отображает структуру каталогов диска.
	DriveComboBox (список дисков)	Win 3.1	Выпадающий список доступных дисков.
	FilterComboBox (список фильтров)	Win 3.1	Выпадающий список фильтров для поиска файлов.
	CDirectoryOutline (дерево каталогов)	Samples	Пример компонента, используемого для отображения структуры каталогов выбранного диска.

Последние пять компонентов в таблице 3.6 являются не законченными диалогами, а их фрагментами, позволяющими строить свои собственные диалоговые окна.

Все диалоги являются невизуальными компонентами, так что место их размещения на форме не имеет значения. При обращении к этим компонентам вызываются стандартные диалоги, вид которых зависит от версии Windows и настройки системы. Так что при запуске одного и того же приложения на компьютерах с разными системами диалоги будут выглядеть по-разному. Например, при русифицированной версии Windows все их надписи будут русскими, а при англоязычной версии надписи будут на английском языке.

Основной метод, которым производится обращение к любому диалогу, — **Execute**. Эта функция открывает диалоговое окно и, если пользователь произвел в нем какой-то выбор, то функция возвращает **true**. При этом в свойствах компонента — диалога запоминается выбор пользователя, который можно прочитать и использовать в дальнейших операциях. Если же пользователь в диалоге нажал кнопку Отмена или клавишу Esc, то функция **Execute** возвращает **false**. Поэтому стандартное обращение к диалогу имеет вид:

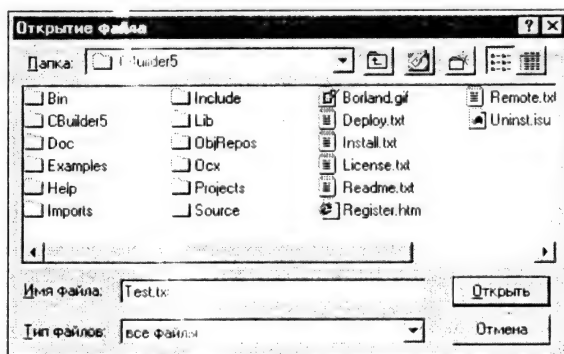
```
if (<имя компонента - диалога> -> Execute())
    <оператор, использующий выбор пользователя>;
```

3.8.2 Диалоги открытия и сохранения файлов — компоненты **OpenDialog**, **SaveDialog**, **OpenPictureDialog**, **SavePictureDialog**

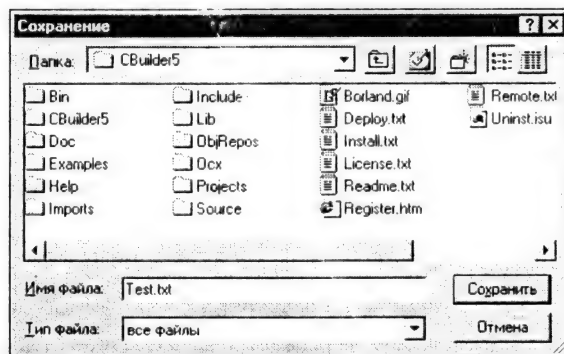
Компоненты **OpenDialog** — диалог «Открыть файл» и **SaveDialog** — диалог «Сохранить файл как...», пожалуй, используются чаще всего, в большинстве приложений. Примеры открываемых ими диалоговых окон приведены на рис. 3.49 и 3.50.

Рис. 3.49

Диалоговое окно открытия файла

**Рис. 3.50**

Диалоговое окно сохранения файла

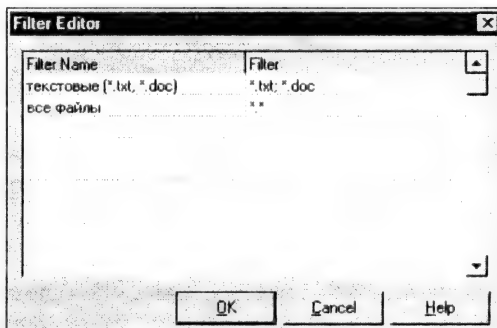


Все свойства этих компонентов одинаковы, только их смысл несколько различен для открытия и сохранения файлов. Основное свойство, в котором возвращается в виде строки выбранный пользователем файл, — **FileName**. Значение этого свойства можно задать и перед обращением к диалогу. Тогда оно появится в диалоге как значение по умолчанию в окне Имя файла (см. рис. 3.49, 3.50).

Типы искоемых файлов, появляющиеся в диалоге в выпадающем списке Тип файла (рис. 3.49, 3.50), задаются свойством **Filter**. В процессе проектирования это свойство проще всего задать с помощью редактора фильтров, который вызывается нажатием кнопки с многоточием около имени этого свойства в Инспекторе Объектов. При этом открывается окно редактора, вид которого представлен на рис. 3.51. В его левой панели Filter Name вы записываете тот текст, который увидит пользователь в выпадающем списке Тип файла диалога. А в правой панели Filter записываются разделенные точками с запятой шаблоны фильтра. В примере рис. 3.51 задано два фильтра: текстовых файлов с расширениями **.txt** и **.doc** и любых файлов с шаблоном ***.*.***.

Рис. 3.51

Окно редактора фильтров



После выхода из окна редактирования фильтров заданные вами шаблоны появятся в свойстве **Filter** в виде строки вида:

```
текстовые (*.txt, *.doc)|*.txt; *.doc|все файлы|*.*
```

В этой строке тексты и шаблоны разделяются вертикальными линиями. В аналогичном виде, если требуется, можно задавать свойство **Filter** программно во время выполнения приложения.

Свойство **FilterIndex** определяет номер фильтра, который будет по умолчанию показан пользователю в момент открытия диалога. Например, значение **FilterIndex** = 1 задает по умолчанию первый фильтр.

Свойство **InitialDir** определяет начальный каталог, который будет открыт в момент начала работы пользователя с диалогом. Если значение этого свойства не задано, то открывается текущий каталог или тот, который был открыт при последнем обращении пользователя к соответствующему диалогу в процессе выполнения данного приложения.

Свойство **DefaultExt** определяет значение расширения файла по умолчанию. Если значение этого свойства не задано, пользователь должен указать в диалоге полное имя файла с расширением. Если же задать значение **DefaultExt**, то пользователь может писать в диалоге имя без расширения. В этом случае будет принято заданное расширение.

Свойство **Title** позволяет вам задать заголовок диалогового окна. Если это свойство не задано, окно открывается с заголовком, определенным в системе (например, «Открытие файла» в окне на рис. 3.49). Но вы можете задать и свой заголовок, подсказывающий пользователю ожидаемые действия. Например, «Укажите имя открываемого файла».

Свойство **Options** определяет условия выбора файла. Множество опций, которые вы можете установить программно или во время проектирования, включает:

ofAllowMultiSelect	Позволяет пользователю выбирать несколько файлов
ofCreatePrompt	В случае, если пользователь написал имя несуществующего файла, появляется замечание и запрос, надо ли создать файл с заданным именем
ofEnableIncludeNotify	Разрешает посылать в диалог сообщения
ofEnableSizing	Разрешает пользователю изменять размер диалогового окна
ofExtensionDifferent	Этот флаг, который можно прочесть после выполнения диалога, показывает, что расширение файла, выбранного пользователем, отличается от DefaultExt
ofFileMustExist	В случае, если пользователь написал имя несуществующего файла, появляется сообщение об ошибке
ofHideReadOnly	Удаляет из диалога индикатор Открыть только для чтения
ofNoChangeDir	После щелчка пользователя на кнопке ОК восстанавливает текущий каталог, независимо от того, какой каталог был открыт при поиске файла
ofNoDereferenceLinks	Запрещает переназначать клавиши быстрого доступа в диалоговом окне
ofNoLongNames	Отображаются только не более 8 символов имени и трех символов расширения
ofNoNetworkButton	Убирает из диалогового окна кнопку поиска в сети. Действует только если флаг ofOldStyleDialog включен

ofNoReadOnlyReturn	Если пользователь выбрал файл только для чтения, то генерируется сообщение об ошибке
ofNoTestFileCreate	Запрещает выбор в сети защищенных файлов и не доступных дисков при сохранении файла
ofNoValidate	Не позволяет писать в именах файлов неразрешенные символы, но не мешает выбирать файлы с неразрешенными символами
ofOldStyleDialog	Создает диалог выбора файла в старом стиле (см. рис. 3.52)
ofOverwritePrompt	В случае, если при сохранении файла пользователь написал имя существующего файла, появляется замечание, что файл с таким именем существует, и запрашивается желание пользователя переписать существующий файл
ofPathMustExist	Генерирует сообщение об ошибке, если пользователь указал в имени файла несуществующий каталог
ofReadOnly	По умолчанию устанавливает индикатор Открыть только для чтения при открытии диалога
ofShareAware	Игнорирует ошибки нарушения условий коллективного доступа и разрешает, несмотря на них, производить выбор файла
ofShowHelp	Отображает в диалоговом окне кнопку Справка

По умолчанию все перечисленные опции, кроме **ofHideReadOnly**, выключены. Но, как видно из их описания, многие из них полезно включить перед вызовом диалогов.

Если вы разрешаете с помощью опции **ofAllowMultiSelect** множественный выбор файлов, то список выбранных файлов можно прочитать в свойстве **Files** типа **TStrings**.

В приведенной таблице даны опции, используемые в 32-разрядных версиях C++Builder. При включении опции **ofOldStyleDialog** диалоговое окно имеет вид, представленный на рис. 3.52. В примере рис. 3.52 диалог открыт с заданным значением свойства **Title** и заданный текст отображается в заголовке окна. Кроме того, в этом примере выключена опция **ofHideReadOnly**, что привело к появлению индикатора Только чтение.

В компонентах диалогов открытия и сохранения файлов предусмотрена возможность обработки ряда событий. Такая обработка может потребоваться, если рассмотренных опций, несмотря на их количество, не хватает, чтобы установить

Рис. 3.52

Диалог в старом стиле при включенной опции **ofOldStyleDialog** и выключенной опции **ofHideReadOnly**.



все диктуемые конкретным приложением ограничения на выбор файлов. Событие **OnCanClose** возникает при нормальном закрытии пользователем диалогового окна после выбора файла. При отказе пользователя от диалога — нажатии кнопки **Отмена**, клавиши **Esc** и т.д. событие **OnCanClose** не наступает. В обработке события **OnCanClose** вы можете произвести дополнительные проверки выбранного пользователем файла и, если по условиям вашей задачи этот выбор недопустим, вы можете известить об этом пользователя и задать значение **false** передаваемому в обработчик параметру **CanClose**. Это не позволит пользователю закрыть диалоговое окно.

Можно также написать обработчики событий **OnFolderChange** — изменение каталога, **OnSelectionChange** — изменение имени файла, **OnTypeChange** — изменение типа файла. В этих обработчиках вы можете предусмотреть какие-то сообщения пользователю.

Теперь приведем пример использования диалогов **OpenDialog** и **SaveDialog**. Пусть ваше приложение включает окно редактирования **RichEdit1** (см. раздел 3.2.4), в которое по команде меню **MainMenu** (см. раздел 3.6.1) Открыть вы хотите загружать текстовый файл и после каких-то изменений, сделанных пользователем, — сохранять по команде Сохранить текст в том же файле, а по команде Сохранить как — в файле с другим именем.

Введите на форму компоненты — диалоги **OpenDialog** и **SaveDialog**. Предположим, что вы оставили их имена по умолчанию — **OpenDialog1** и **SaveDialog1**. Поскольку после чтения файла вам надо запомнить его имя, чтобы знать под каким именем потом его сохранять, вы можете определить для этого имени переменную, назвав ее, например, **MyFName**. Для этого в модуле формы объявите эту глобальную переменную:

```
AnsiString MyFName = "";
```

Тогда обработка команды Открыть может сводиться к следующему оператору:

```
if (OpenDialog1->Execute())
{
    MyFName = OpenDialog1->FileName;
    RichEdit1->Lines->LoadFromFile(OpenDialog1->FileName);
}
```

Этот оператор вызывает диалог, проверяет, выбрал ли пользователь файл (если выбрал, то функция **Execute** возвращает **true**), после чего имя выбранного файла (**OpenDialog1->FileName**) сохраняется в переменной **MyFName** и файл загружается в текст **RichEdit1** методом **LoadFromFile**.

Обработка команды Сохранить как выполняется операторами:

```
SaveDialog1->FileName = MyFName;
if (SaveDialog1->Execute())
{
    MyFName = SaveDialog1->FileName;
    RichEdit1->Lines->SaveToFile(SaveDialog1->FileName);
}
```

Первый из этих операторов присваивает свойству **FileName** компонента **SaveDialog1** запомненное имя файла. Это имя по умолчанию будет предложено пользователю при открытии диалога Сохранить как. Следующий оператор открывает диалог и, если пользователь выбрал в нем файл, запоминает новое имя файла и сохраняет в файле с этим именем текст компонента **RichEdit1**.

Обработка команды Сохранить выполняется операторами

```
if(MyFName != "")
    RichEdit1->Lines->SaveToFile(MyFName);
else
    if (SaveDialog1->Execute())
    {
```

```
MyFName = SaveDialog1->FileName;
RichEdit1->Lines->SaveToFile(SaveDialog1->FileName);
}
```

Если имя файла **MyFName** не равно пустой строке, т.е. известно, то нет необходимости обращаться к какому-то диалогу. Текст сохраняется методом **SaveToFile**. Если же имя файла неизвестно, то текст сохраняется с помощью диалога **SaveDialog1** так же, как было рассмотрено выше.

Мы рассмотрели диалоги открытия и сохранения файлов произвольного типа. В библиотеке C++Builder 5 имеются также специализированные диалоги открытия и сохранения графических файлов: **OpenPictureDialog** и **SavePictureDialog**. Пример окна, открываемого этими компонентами вы можете увидеть, например, на рис. 5.2 в разделе 5.1.1.1 главы 5. От окон, открываемых компонентами **OpenDialog** и **SaveDialog** (рис. 3.49, 3.50), они отличаются удобной возможностью просматривать изображения в процессе выбора файла.

Свойства компонентов **OpenPictureDialog** и **SavePictureDialog** ничем не отличаются от свойств компонентов **OpenDialog** и **SaveDialog**. Единственное отличие — заданное значение по умолчанию свойства **Filter** в **OpenPictureDialog** и **SavePictureDialog**. В этих компонентах заданы следующие фильтры:

All (*.jpg;*.jpeg;*.bmp;*.ico;*.emf;*.wmf)	*.jpg;*.jpeg;*.bmp;*.ico;*.emf;*.wmf
JPEG Image File (*.jpg)	*.jpg
JPEG Image File (*.jpeg)	*.jpeg
Bitmaps (*.bmp)	*.bmp
Icons (*.ico)	*.ico
Enhanced Metafiles (*.emf)	*.emf
Metafiles (*.wmf)	*.wmf

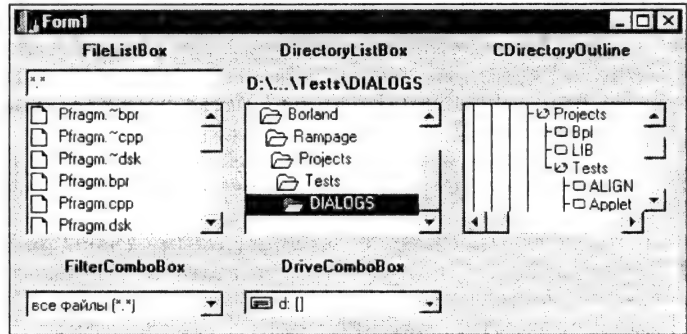
В этих фильтрах перечислены все типы графических файлов, с которыми может работать диалог. Так что вам остается удалить, если хотите, фильтры тех файлов, с которыми вы не хотите работать, добавить, может быть, фильтр «Все файлы (*.*)» и перевести на русский язык названия типов.

3.8.3 Фрагменты диалогов — компоненты **DriveComboBox**, **DirectoryListBox**, **FilterComboBox**, **FileListBox** и **CDirectoryOutline**

Помимо законченных диалогов работы с файлами, в C++Builder имеется ряд компонентов, представляющих собой фрагменты диалогов: выпадающие списки дисков (драйверов) — **DriveComboBox** и фильтров (масок) файлов — **FilterComboBox**, списки каталогов — **DirectoryListBox** и файлов — **FileListBox**, дерево каталогов — **CDirectoryOutline**. Внешний вид этих компонентов вы можете увидеть на рис. 3.53. Компоненты работы с файловой системой облегчают вам создание собственных диалоговых окон, что нередко требуется. Например, вы можете захотеть включить в диалоговое окно отображение каких-то характеристик файлов (размера, даты создания и т.п.) или оперативный просмотр содержания текстовых файлов. Тогда вам очень пригодятся готовые компоненты работы с файлами. Правда, все они, кроме **CDirectoryOutline**, расположены на странице Win 3.1 палитры компонентов. Это значит, что они не рекомендуются для 32-разрядных приложений. Но, во-первых, у вас остается компонент **CDirectoryOutline**. Кроме того,

Рис. 3.53

Компоненты — фрагменты диалогов



никто не запрещает вам все-таки использовать и остальные компоненты в любых приложениях C++Builder. И, наконец, если уж вы хотите четко следовать рекомендации не использовать первые четыре фрагмента диалогов в 32-разрядных приложениях, вы можете разработать свои аналогичные компоненты, используя обычный компонент **ComboBox**.

Начнем рассмотрение компонентов работы с файловой системой с компонента **DriveComboBox** — выпадающего списка дисков (драйверов). При размещении на форме этот компонент автоматически отображает список имеющихся на компьютере дисков. Во время выполнения приложения вы можете прочитать имя выбранного пользователем диска в свойстве **Drive**, а строку, содержащуюся в окне списка — в свойстве **Text**.

Свойство **TextCase** задает регистр отображения: **tcUpperCase** — в верхнем регистре, **tcLowerCase** — в нижнем.

Связать компонент **DriveComboBox** со списком каталогов, отображаемых компонентом **DirectoryListBox**, можно во время проектирования через свойство **DirList** компонента **DriveComboBox**. Это свойство может указывать на компонент типа **DirectoryListBox**. Можно обеспечить связь этих двух типов компонентов и программно, включив в обработчик события **OnChange** компонента **DriveComboBox** оператор

```
DirectoryListBox1->Drive = DriveComboBox1->Drive;
```

Этот оператор задает имя диска, выбранное пользователем в компоненте **DriveComboBox1**, свойству **Drive** списка каталогов **DirectoryListBox1**.

Аналогичным оператором можно обеспечить связь компонента **DriveComboBox** с деревом каталогов и файлов в компоненте **CDirectoryOutline**:

```
CDirectoryOutline1->Drive = DriveComboBox1->Drive;
```

Рассмотрим теперь выпадающий список фильтров — компонент **FilterComboBox**. Его основное свойство — **Filter**, которое задается так же, как в описанных ранее диалогах. К отдельным частям фильтра — тексту и маске, можно получить доступ через свойства **Text** и **Mask** соответственно. Связь компонента со списком файлов типа **TFileListBox** можно установить, задав свойство **FileList**.

Компонент **DirectoryListBox** отображает список каталогов диска, заданного свойством **Drive**. Значение этого свойства можно установить программно во время выполнения. Как уже говорилось выше, связь этого свойства с выбранным пользователем диском в компоненте **DriveComboBox** устанавливается или программно, или с помощью свойства **DirectoryListBox** компонента **DriveComboBox**.

Связь списка каталогов с компонентом типа **TFileListBox**, отображающим список файлов, осуществляется с помощью свойства **FileList**. Можно также использовать результаты выбора пользователем каталога, читая свойство **Directory** в обработчике события **OnChange**.

С компонентом **DirectoryListBox** можно также связать метку типа **Label**. В этой метке будет отображаться путь к текущему каталогу. Если путь не умещается в метке, он автоматически отображается в сокращенном виде (см. рис. 3.53) с помощью функции **MinimizeName**. Метка, отображающая каталог, указывается в свойстве **DirLabel**.

Список файлов содержится в компоненте **FileListBox**. Его свойства **Drive**, **Directory** и **Mask** определяют соответственно диск, каталог и маску файлов. Эти свойства можно устанавливать программно или связывая описанным ранее способом компонент **FileListBox** с компонентами **DriveComboBox**, **DirectoryListBox** и **FilterComboBox**. Свойство **FileType** позволяет включать в список не все файлы, а только те, которые имеют соответствующие атрибуты. Свойство **FileType** представляет собой множество, указывающее типы включаемых файлов. Элементы этого множества могут иметь значения: **ftReadOnly** — только для чтения, **ftHidden** — невидимые, **ftSystem** — системные, **ftVolumeID** — обозначения дисков, **ftDirectory** — каталоги, **ftArchive** — архивные, **ftNormal** — не имеющие особых атрибутов.

Свойство **ShowGlyphs** разрешает или исключает показ пиктограмм файлов (в примере рис. 3.53 это свойство = **true**).

Свойство **MultiSelect** разрешает выбор нескольких файлов.

Основное свойство, в котором можно прочесть имя выбранного пользователем файла — **FileName**.

Со списком файлов может быть связано окно редактирования **Edit**, в котором отображается выбранный файл (см. окно над списком файлов на рис. 3.53). На этот список указывает устанавливаемое во время проектирования свойство **FileEdit**.

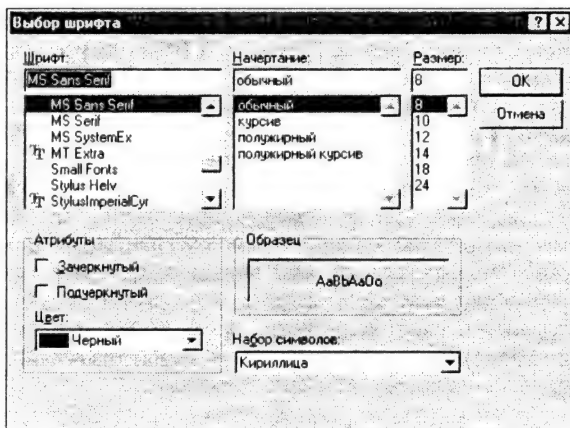
Теперь рассмотрим компонент **CDirectoryOutline**, содержащий дерево каталогов. В этом компоненте значение диска устанавливается свойством **Drive**. Текущий каталог, выбранный пользователем, можно прочесть в свойстве **Directory**. Свойство **TextCase** определяет стиль отображения имен каталогов: **tcLowerCase** — преобразование к нижнему регистру, **tcUpperCase** — к верхнему, **tcAsIs** — без преобразования (этот режим использован в примере рис. 3.53). Остальные свойства идентичны компоненту **OutLine**, на основе которого построен данный пример. Вы можете найти исходный текст этого примера в каталоге ...\\source\\samples.

3.8.4 Диалог выбора шрифта — компонент **FontDialog**

Компонент **FontDialog** вызывает диалоговое окно выбора атрибутов шрифта, представленное на рис. 3.54. В нем пользователь может выбрать имя шрифта, его стиль (начертание), размер и другие атрибуты.

Рис. 3.54

Диалоговое окно выбора атрибутов шрифта



Основное свойство компонента — **Font** типа **TFont** (см. разделы 4.1.5 и 16.4), в котором вы можете задать при желании начальные установки атрибутов шрифта и в котором вы можете прочесть значения атрибутов, выбранные пользователем в процессе диалога.

Свойства **MaxFontSize** и **MinFontSize** устанавливают ограничения на максимальный и минимальный размеры шрифта. Если значения этих свойств равны 0 (по умолчанию), то никакие ограничения на размер не накладываются. Если же значения свойств заданы (обычно это целесообразно делать исходя из размеров компонента приложения, для которого выбирается шрифт), то в списке Размер диалогового окна (см. рис. 3.54) появляются только размеры, укладывающиеся в заданный диапазон. При попытке пользователя задать недопустимый размер ему будет выдано предупреждение вида «Размер должен лежать в интервале ...» и выбор пользователя отменится. Свойства **MaxFontSize** и **MinFontSize** действуют только при включенной опции **fdLimitSize** (см. ниже).

Свойство **Device** определяет, из какого списка возможных шрифтов будет предложен выбор в диалоговом окне: **fdScreen** — из списка экрана (по умолчанию), **fdPrinter** — из списка принтера, **fdBoth** — из обоих.

Свойство **Options** содержит множество опций:

fdAnsiOnly	Отображать только множество шрифтов символов Windows, не отображать шрифтов со специальными символами
fdApplyButton	Отображать в диалоге кнопку Применить независимо от того, предусмотрен ли обработчик события OnApply
fdEffects	Отображать в диалоге индикаторы специальных эффектов (подчеркивание и др.) и список Цвет
fdFixedPitchOnly	Отображать только шрифты с постоянной шириной символов
fdForceFontExist	Позволять пользователю выбирать шрифты только из списка, запрещать ему вводить другие имена
fdLimitSize	Разрешить использовать свойства MaxFontSize и MinFontSize , ограничивающие размеры шрифта
fdNoFaceSel	Открывать диалоговое окно без предварительно установленного имени шрифта
fdNoOEMFonts	Удалять из списка шрифтов шрифты OEM
fdScalableOnly	Отображать только масштабируемые шрифты, удалять из списка не масштабируемые (шрифты bitmap)
fdNoSimulations	Отображать только шрифты и их начертания, напрямую поддерживаемые файлами, не отображая шрифты, в которых жирный стиль и курсив синтезируется
fdNoSizeSel	Открывать диалоговое окно без предварительно установленного размера шрифта
fdNoStyleSel	Открывать диалоговое окно без предварительно установленного начертания шрифта
fdNoVectorFonts	Удалять из списка векторные шрифты (типа Roman или Script для Windows 1.0)
fdShowHelp	Отображать в диалоговом окне кнопку Справка
fdTrueTypeOnly	Предлагать в списке только шрифты TrueType
fdWysiwyg	Предлагать в списке только шрифты, доступные и для экрана, и для принтера, удаляя из него аппаратно зависимые шрифты

По умолчанию все эти опции, кроме **fdEffects**, отключены.

Если установить опцию **fdApplyButton**, то при нажатии пользователем кнопки Применить возникает событие **OnApply**, в обработчике которого вы можете написать код, который применит выбранные пользователем атрибуты, не закрывая диалогового окна.

Приведем примеры применения компонента **FontDialog**. Пусть ваше приложение включает окно редактирования **Memo1**, шрифт в котором пользователь может выбирать командой меню Шрифт. Вы ввели в приложение компонент **FontDialog**, имя которого по умолчанию **FontDialog1**. Тогда обработчик команды Шрифт может иметь вид:

```
if (FontDialog1->Execute())
    Memo1->Font->Assign(FontDialog1->Font);
```

Приведенный оператор вызывает диалог выбора атрибутов шрифта и, если пользователь произвел выбор, то значения всех выбранных атрибутов, содержащиеся в свойстве **FontDialog1->Font**, присваиваются атрибутам окна редактирования, содержащимся в свойстве **Memo1.Font**. Шрифт в окне редактирования немедленно изменится.

Если вы установите в компоненте **FontDialog1** опцию **fdApplyButton**, то можете написать обработчик события **OnApply**:

```
Memo1->Font->Assign(FontDialog1->Font);
```

Тогда пользователь может наблюдать изменения в окне **Memo1**, нажимая в диалоговом окне кнопку Применить и не прерывая диалога. Это очень удобно, так как позволяет пользователю правильно подобрать атрибуты шрифта.

Если в качестве окна редактирования в вашем приложении вы используете **RichEdit**, то можете предоставить пользователю выбирать атрибуты шрифта для выделенного фрагмента текста или для вновь вводимого текста. Тогда выполнение команды меню Шрифт может осуществляться операторами:

```
if (FontDialog1->Execute())
    RichEdit1->SelAttributes->Assign(FontDialog1->Font);
```

Вы можете разрешить пользователю изменять шрифт не только отдельных компонентов, но и всех компонентов и надписей на форме. Это осуществляется оператором:

```
if (FontDialog1->Execute())
    Font->Assign(FontDialog1->Font);
```

В этом операторе свойство **Font** без ссылки на компонент подразумевает шрифт формы.

3.8.5 Диалог выбора цвета — компонент **ColorDialog**

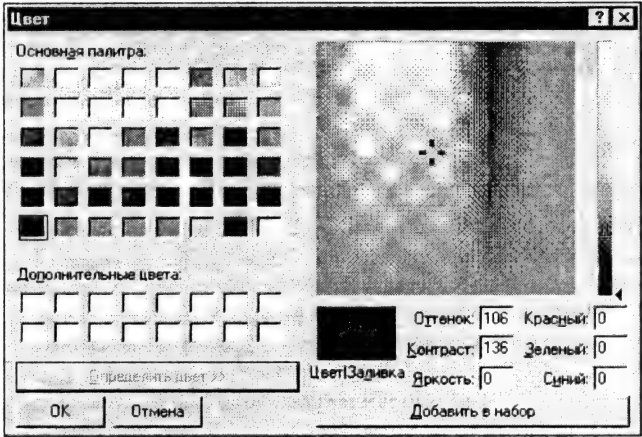
Компонент **ColorDialog** вызывает диалоговое окно выбора цвета, представленное на рис. 3.55. В нем пользователь может выбрать цвет из базовой палитры или, нажав кнопку Определить цвет, раскрыть дополнительную панель (на рис. 3.55 она раскрыта), позволяющую синтезировать цвет, отличный от базовых. Синтезированный цвет можно добавить кнопкой Добавить в набор в палитру дополнительных цветов на левой панели и использовать его в дальнейшем.

Основное свойство компонента **ColorDialog** — **Color**. Это свойство соответствует тому цвету, который выбрал в диалоге пользователь. Если при вызове диалога желательно установить некоторое начальное приближение цвета, это можно сделать, установив **Color** предварительно во время проектирования или программно.

Свойство **CustomColors** типа **TStrings** позволяет задать заказные цвета дополнительной палитры. Каждый цвет определяется строкой вида

```
<Имя цвета>=<шестнадцатеричное представление цвета>;
```

Рис. 3.55
Диалоговое окно выбора цвета



Имена цветов задаются от **ColorA** (первый цвет) до **ColorP** (шестнадцатый, последний). Например, строка

```
ColorA=808022
```

задает первый заказной цвет. Подробнее о задании цветов см. в справочной части книги в главе 16.

Свойство **Options** содержит множество следующих опций:

cdFullOpen	Отображать сразу при открытии диалогового окна панели определения заказных цветов
cdPreventFullOpen	Запретить появление в диалоговом окне кнопки Определить цвет, так что пользователь не сможет определять новые цвета
cdShowHelp	Добавить в диалоговое окно кнопку Справка
cdSolidColor	Указать Windows использовать сплошной цвет, ближайший к выбранному (это обедняет палитру)
cdAnyColor	Разрешать пользователю выбирать любые не сплошные цвета (такие цвета могут быть не ровными)

По умолчанию все опции выключены.

Приведем пример применения компонента **ColorDialog**. Если вы хотите, чтобы пользователь мог задать цвет какого-то объекта, например, цвет фона компонента **Memo1**, то это можно реализовать оператором

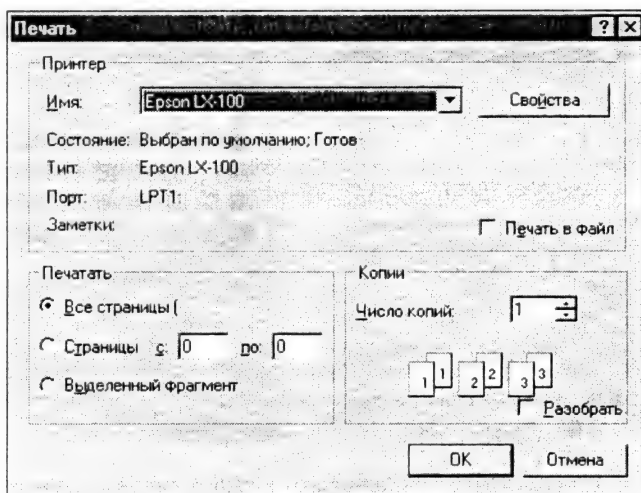
```
if (ColorDialog1->Execute())  
    Memo1->Color = ColorDialog1->Color;
```

3.8.6 Диалоги печати и установки принтера — компоненты **PrintDialog** и **PrinterSetupDialog**

Компонент **PrintDialog** вызывает диалоговое окно печати, представленное на рис. 3.56. В нем пользователь может выбрать принтер и установить его свойства, указать число копий и последовательность их печати, печатать в файл или непосредственно на принтер, выбрать печатаемые страницы или печатать только выделенный фрагмент.

Рис. 3.56

Диалоговое окно настройки печати



Компонент **PrintDialog** не осуществляет печать. Он только позволяет пользователю задать атрибуты печати. А сама печать должна осуществляться программно с помощью объекта **Printer** или иным путем (о способах печати см. раздел 4.6).

Рассмотренные ранее диалоговые компоненты возвращали одно свойство — имя файла, цвет, или один объект — **Font**, содержащий множество свойств. В отличие от них компонент **PrintDialog** возвращает ряд свойств, характеризующих выбранные пользователем установки. Это следующие свойства:

PrintRange	Показывает выбранную пользователем радиокнопку из группы Печать: prAllPages — выбрана кнопка Все страницы, prSelection — выбрана кнопка Страницы с ... по ..., prPageNums — выбрана кнопка Страницы
FromPage	Показывает установленную пользователем начальную страницу в окне Страницы с ... по ...
ToPage	Показывает установленную пользователем конечную страницу в окне Страницы с ... по ...
PrintToFile	Показывает, выбран ли пользователем индикатор Печать в файл
Copies	Показывает установленное пользователем число копий
Collate	Показывает, выбран ли пользователем индикатор Разобрать

Перед вызовом диалога желательно определить, сколько страниц в печатаемом тексте, и задать параметры **MaxPage** и **MinPage** — максимальный и минимальный номера страниц. В противном случае пользователю в диалоговом окне не будет доступна кнопка Страницы с ... по Кроме того следует определить множество опций в свойстве **Options**:

poDisablePrintToFile	Запретить доступ к индикатору Печать в файл. Эта опция работает только при включенной опции poPrintToFile
poHelp	Отображать в диалоговом окне кнопку Справка. Опция может не работать для некоторых версий Windows 95/98
poPageNums	Сделать доступной радиокнопку Страницы, позволяющую пользователю задавать диапазон печатаемых страниц

poPrintToFile	Отображать в диалоговом окне кнопку Печать в файл
poSelection	Сделать доступной кнопку Выделение, позволяющую пользователю печатать только выделенный текст
poWarning	Выдавать замечания, если пользователь пытается послать задачу на неустановленный принтер

Теперь остановимся на компоненте **PrinterSetupDialog**, вызывающем диалоговое окно установки принтера. Это единственный диалоговый компонент, не имеющий никаких специфических свойств, которые надо было бы устанавливать или читать. Диалог выполняет операции по установке принтера, на котором будет производиться печать, и задании его свойств. Этот диалог не возвращает никаких параметров.

3.8.7 Диалоги поиска и замены текста — компоненты FindDialog и ReplaceDialog

Компоненты **FindDialog** и **ReplaceDialog**, вызывающие диалоги поиска и замены фрагментов текста (рис. 3.57 и 3.58), очень похожи и имеют одинаковые свойства, кроме одного, задающего заменяющий текст в компоненте **ReplaceDialog**. Такое сходство не удивительно, поскольку **ReplaceDialog** — производный класс от **FindDialog**.

Рис. 3.57
Диалоговое окно поиска фрагмента текста

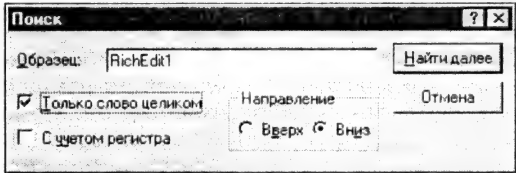
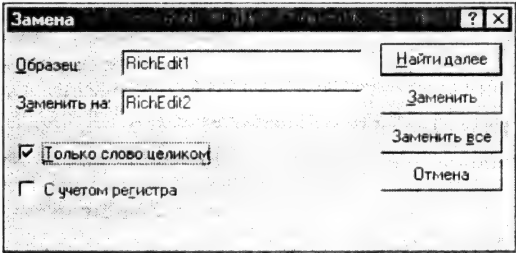


Рис. 3.58
Диалоговое окно замены фрагмента текста



Компоненты имеют следующие основные свойства:

FindText	Текст, заданный пользователем для поиска или замены. Программно может быть установлен как начальное значение, предлагаемое пользователю
ReplaceText	Только в компоненте ReplaceDialog — текст, который должен заменять FindText
Position	Позиция левого верхнего угла диалогового окна, заданная типом TPoint — записью, содержащей поля X (экранный координата по горизонтали) и Y (экранный координата по вертикали)
Left	Координата левого края диалогового окна, то же, что Position.X
Top	Координата верхнего края диалогового окна, то же, что Position.Y
Options	Множество опций

Последний параметр **Options** — может содержать следующие свойства:

frDisableMatchCase	Делает недоступным индикатор С учетом регистра в диалоговом окне
frDisableUpDown	Делает недоступными в диалоговом окне кнопки Вверх и Вниз группы Направление, определяющие направление поиска
frDisableWholeWord	Делает недоступным индикатор Только слово целиком в диалоговом окне
frDown	Выбирает кнопку Вниз группы Направление при открытии диалогового окна. Если эта опция не установлена, то выбирается кнопка Вверх
frFindNext	Эта опция включается автоматически, когда пользователь в диалоговом окне щелкает на кнопке Найти далее, и выключается при закрытии диалога
frHideMatchCase	Удаляет индикатор С учетом регистра из диалогового окна
frHideWholeWord	Удаляет индикатор Только слово целиком из диалогового окна
frHideUpDown	Удаляет кнопки Вверх и Вниз из диалогового окна
frMatchCase	Этот флаг включается и выключается, если пользователь включает и выключает опцию С учетом регистра в диалоговом окне. Можно установить эту опцию по умолчанию во время проектирования, чтобы при открытии диалога она была включена
frReplace	Применяется только для ReplaceDialog . Этот флаг устанавливается системой, чтобы показать, что текущее (и только текущее) найденное значение FindText должно быть заменено значением ReplaceText
frReplaceAll	Применяется только для ReplaceDialog . Этот флаг устанавливается системой, чтобы показать, что все найденные значения FindText должны быть заменены значениями ReplaceText
frShowHelp	Задаёт отображение кнопки Справка в диалоговом окне
frWholeWord	Этот флаг включается и выключается, если пользователь включает и выключает опцию Только слово целиком в диалоговом окне. Можно установить эту опцию по умолчанию во время проектирования, чтобы при открытии диалога она была включена

Сами по себе компоненты **FindDialog** и **ReplaceDialog** не осуществляют ни поиска, ни замены. Они только обеспечивают интерфейс с пользователем. А поиск и замену надо осуществлять программно. Для этого можно пользоваться событием **OnFind**, происходящим, когда пользователь нажал в диалоге кнопку Найти далее, и событием **OnReplace**, возникающим, если пользователь нажал кнопку Заменить или Заменить все. В событии **OnReplace** узнать, какую именно кнопку нажал пользователь, можно по значениям флагов **frReplace** и **frReplaceAll**.

Поиск заданного фрагмента в компоненте **RichEdit** легко проводить, используя его метод **FindText**, объявленный следующим образом:


```
int __fastcall FindText(const System::AnsiString SearchStr,
                       int StartPos, int Length,
                       TSearchTypes Options);
```

Этот метод ищет в тексте **RichEdit** фрагмент, заданный параметром **SearchStr**. Поиск идет начиная с позиции **StartPos** (позиция первого символа текста считается нулевой) на протяжении **Length** символов. Параметр **Options** является множеством, которое может содержать элементы **stWholeWord** (поиск только целого слова) и **stMatchCase** (поиск с учетом регистра). Метод возвращает позицию найденного вхождения. Если заданный фрагмент не найден, возвращается -1.

Ниже приведен код, осуществляющий поиск заданного фрагмента в тексте компонента **RichEdit1**.

```
void __fastcall TForm1::MFindClick(TObject *Sender)
{
    /* начальное значение текста поиска - текст, выделенный в      Mem1 */
    FindDialog1->FindText = Mem1->SelText;
    FindDialog1->Execute();
}
//-----

void __fastcall TForm1::FindDialog1Find(TObject *Sender)
{
    int FoundAt, StartPos, ToEnd;

    /* если было выделение, то поиск идет начиная с его
       последнего символа, иначе - с позиции курсора */
    StartPos = Mem1->SelStart;
    if (Mem1->SelLength)
        StartPos += Mem1->SelLength;

    /* ToEnd - длина текста, начиная с первой позиции поиска
       и до конца */
    ToEnd = Mem1->Text.Length() - StartPos;

    /* поиск с учетом или без учета регистра в зависимости от
       установки пользователя */
    if (FindDialog1->Options.Contains(frMatchCase))
        FoundAt = StartPos +
            Mem1->Text.SubString(StartPos+1,
                                ToEnd).Pos(FindDialog1->FindText);
    else
        FoundAt = StartPos +
            Mem1->Text.SubString(StartPos+1,
                                ToEnd).LowerCase().Pos(FindDialog1->FindText.LowerCase());

    if (FoundAt != StartPos) // если найдено
    {
        Mem1->SetFocus();
        Mem1->SelStart = FoundAt-1;
        Mem1->SelLength = FindDialog1->FindText.Length();
    }
    else ShowMessage("Текст '" + FindDialog1->FindText +
                     "' не найден");
}
```

Функция **MFindClick** задает в качестве начального значения для поиска текст, выделенный в **RichEdit1**, и затем вызывает диалог поиска **FindDialog1**. Функция **FindDialog1Find** является обработчиком события **OnFind** компонента **FindDialog1**. Она срабатывает, когда пользователь нажал в диалоге кнопку Найти далее. Комментарии в тексте этой функции поясняют этапы поиска. Сначала про-


```

else
    FoundAt = StartPos +
        Memol->Text.SubString(StartPos+1,
            ToEnd).LowerCase().Pos(FindDialog1->FindText.LowerCase());

if (FoundAt != StartPos) // если найдено
{
    Memol->SetFocus();
    Memol->SelStart = FoundAt-1;
    Memol->SelLength = FindDialog1->FindText.Length();
}
else ShowMessage("Текст '" + FindDialog1->FindText +
    "' не найден");
}

```

Программа аналогична приведенной ранее для компонента **RichEdit** и отличается только несколькими операторами, осуществляющими непосредственно поиск.

При реализации команды **Заменить** приведенные выше процедуры можно оставить теми же самыми, заменив в них **FindDialog1** на **ReplaceDialog1**. Дополнительно можно написать процедуру обработки события **OnReplace** компонента **ReplaceDialog1**. Кроме того желательно обеспечить, чтобы при нажатии пользователем в диалоге клавиши **Заменить** все программа просматривала бы весь текст и проводила все замены без дополнительных вопросов пользователю. Для этого можно в конце обработчика события **OnFind** вставить оператор, который в случае, если пользователь нажал в диалоге клавишу **Заменить** все (см. рис. 3.58), вызывал бы обработчик события **OnReplace**. В итоге текст, обеспечивающий замену в компоненте **RichEdit**, может иметь вид:

```

void __fastcall TForm1::ReplaceDialog1Find(TObject *Sender)
{
    ...
    // Если нажата кнопка "Заменить все", то уход на замену
    if (ReplaceDialog1->Options.Contains(frReplaceAll))
        ReplaceDialog1Replace(Sender);
}
//-----

void __fastcall TForm1::ReplaceDialog1Replace(TObject *Sender)
{
    if (RichEdit1->SelText != "") // Если есть выделенный текст
    {
        // Замена выделенного текста
        RichEdit1->SelText = ReplaceDialog1->ReplaceText;
        // Если нажата кнопка "Заменить все", то изменение позиции
        if (ReplaceDialog1->Options.Contains(frReplaceAll))
            RichEdit1->SelStart += ReplaceDialog1->ReplaceText.Length();
    }
    else if (ReplaceDialog1->Options.Contains(frReplace))
    {
        ShowMessage("Текст '" + ReplaceDialog1->FindText +
            "' не найден");
        return;
    }
    // Если нажата кнопка "Заменить все", то уход на поиск
    if (ReplaceDialog1->Options.Contains(frReplaceAll))
        ReplaceDialog1Find(Sender);
}

```

В функции **ReplaceDialog1Find** в конце поиска очередного вхождения искомого фрагмента в текст проверяется нажатие пользователем кнопки **Заменить** все. Если она нажата, то происходит обращение к функции **ReplaceDialog1Replace**, в

которой осуществляется замена текста, после чего сразу автоматически опять вызывается функция **ReplaceDialog1Find**. Таким образом без остановок просматриваются и заменяются все вхождения искомого фрагмента в текст. Если же кнопка **Заменить все** не нажата, то программа останавливается и ждет, пока пользователь нажмет в диалоге (см. рис. 3.58) кнопку **Найти далее** или **Заменить**. Если нажимается кнопка **Найти далее**, то производится следующий вызов **ReplaceDialog1Find**. А при нажатии кнопки **Заменить** вызывается функция **ReplaceDialog1Replace**, которая заменяет выделенный текст.

Приведенные коды рассчитаны на компонент **RichEdit**. Для компонента **Мемо** в них просто надо заменить **RichEdit1** на **Memo1**.

3.9 Компоненты организации управления приложением

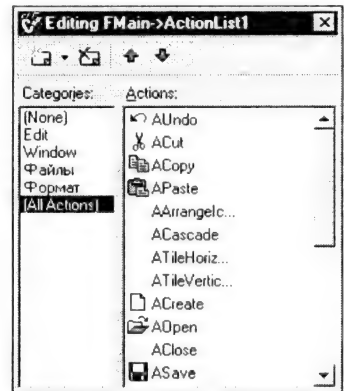
3.9.1 Диспетчеризация событий — компоненты, связанные с **ActionList**

Начиная с C++Builder 4 появился инструментарий, который, не добавляя никаких принципиально новых возможностей, позволяет систематизировать и упорядочить разработку объектно-ориентированных приложений. К тому же грамотное его использование позволяет сэкономить немало времени при проектировании.

В начале проектирования приложения разработчик должен представить себе список тех действий, которые может осуществлять пользователь. Конечно, в процессе проектирования этот список будет пополняться и изменяться, но некоторое начальное приближение очень полезно продумать заранее. Практическая реализация составленного вами списка действий может начинаться с переноса на проектируемую форму невидимого компонента **ActionList**, расположенного на странице библиотеки **Standard**. Сделав на этом компоненте двойной щелчок, вы попадаете в редактор действий (рис. 3.59), позволяющий вводить и упорядочивать действия.

Рис. 3.59

Окно редактора действий



Щелчок правой кнопкой мыши или щелчок на маленькой кнопке со стрелкой вниз правее первой быстрой кнопки окна редактирования позволит вам выбрать одну из команд: **New Action** (новое действие) или **New Standard Action** (новое стандартное действие). Первая из них относится к вводу нового действия любого типа. По умолчанию эти действия будут получать имена **Action1**, **Action2** и т.д. Вторая команда открывает окно, в котором вы можете выбрать необходимое вам стандартное действие (или сразу несколько действий). После этого в правом окне

(Actions) редактора появятся имена выбранных действий, а в левом (Categories) — категории действий.

Каждое действие, которое вы внесли в список — это объект типа **TAction** для нестандартных действий или других производных типов для стандартных. Выбрав в окнах редактора ту или иную категорию или [AllActions] (все категории), а в правом — конкретное действие, вы можете увидеть в Инспекторе Объектов его свойства и события. Вы можете установить свойство **Name** (имя) — оно появится в правом окне редактора свойств и будет в дальнейшем фигурировать в заголовках обработчиков событий. Во избежание путаницы, которая иногда может возникнуть в коде программы, избегайте имен действий, совпадающих с именами каких-то функций, переменных или констант. Удачным, как мне кажется, выходом является добавление к имени каждого действия символа «А», как вы можете видеть это на рис. 3.59.

Вы можете для каждого действия установить надпись (**Caption**), которая далее будет появляться в инициаторах действия — кнопках, разделах меню и т.д. Можно задать быстрые клавиши (**ShortCut**), надписи на ярлычках подсказок и в строке состояний (**Hint**), идентификатор темы контекстной справки (**HelpContext**), свойства **Enabled** (доступность), **Visible** (видимость) и другие обычные для многих компонентов свойства.

Можно для каждого или некоторых действий указать свойство **ImageIndex**, которое является индексом (начиная с 0) изображения, соответствующего данному действию в отдельном компоненте списка изображений **ImageList** (см. раздел 3.9.2). Этот индекс передается в дальнейшем компонентам, связанным с данным событием — разделам меню, кнопкам. Если в свойстве **Images** компонента **ActionList** указать имя списка, размещенного на форме и заполненного изображениями, то эти изображения появятся также в окне редактора действий (рис. 3.59).

Свойство **Category** (категория) не имеет отношения к выполнению приложения. Задание категории просто позволяет в процессе проектирования сгруппировать действия по их назначению. Вы можете для каждого действия выбрать категорию из выпадающего списка, или написать имя новой категории и отнести к ней какие-то действия. Например, на рис. 3.59 видна введенная пользователем категория Файлы, к которой отнесены действия, связанные с меню Файл.

На странице событий Инспектора Объектов для каждого действия определено три события: **OnExecute**, **OnUpdate** и **OnHint**.

Событие **OnExecute** возникает в момент, когда пользователь инициализировал действие, например, щелкнув на компоненте (разделе меню, кнопке), связанном с данным действием. Обработчик этого события должен содержать процедуру, реализующую данное действие. Например, обработчик события **OnExecute** действия **Exit** может в простейшем случае иметь вид

```
void __fastcall TFMain::AExitExecute(TObject *Sender)
{
    Close();
}
```

а в более сложных случаях может содержать проверку возможности закрыть приложение, запросы пользователю и т.д. Одним из преимуществ использования действий является то, что заголовки обработчиков приобретают осмысленный характер и код становится более прозрачным. Действительно, гораздо понятнее заголовки **ExitExecute**, чем, например, **Button7Click** или **N14Click** (попробуйте найти в вашем большом приложении, где эта кнопка **Button7** или раздел меню **N14**). В результате вы избавляетесь от необходимости давать осмысленные имена кнопкам и разделам меню, т.е. облегчаете свою работу с компонентами.

Событие **OnUpdate** периодически возникает в промежутках между действиями. Возникновение этих событий прекращается только во время реализации события или во время, когда пользователь ничего не делает и компьютер находится в

состоянии ожидания действий. Обработчик события **OnUpdate** может содержать какие-то настройки, подготовку ожидаемых дальнейших действий или выполнение каких-то фоновых операций.

Событие **OnHint** возникает в момент, когда на экране отображается ярлычок подсказки в результате того, что пользователь задержал курсор мыши над компонентом, инициализирующим событие.

Наличие в объекте действия событий **OnUpdate** и **OnHint** обогащает ваши возможности по проектированию приложения. Без этого объекта подобные события отсутствуют и их при необходимости приходится моделировать более сложными приемами.

Связь объектов действий с конкретными инициализаторами действий — управляющими элементами типа кнопок, разделов меню и т.д., осуществляется через свойство **Action**, имеющееся у всех управляющих элементов. Поместите на вашу форму кнопку, и вы увидите в Инспекторе Объектов это свойство. Раскройте его выпадающий список и выберите из него действие, которое вами было предварительно описано. Вы сможете заметить, что после этого в кнопку перенесутся такие свойства объекта действия, как **Caption**, **Hint** и др., и что в ее событие **OnClick** подставится обработчик, предусмотренный вами для данного действия. Если далее вы введете в приложение меню с разделом, соответствующим тому же действию, и укажете в нем то же значение свойства **Action**, то свойства и обработчики событий объекта действия будут перенесены и на этот раздел меню. Вам не придется повторно задавать значение **Hint**, **Caption** и др.

Подобная связь между свойствами объекта действия и свойствами управляющих элементов выполняется классом **TActionLink** и его наследниками. Передаются в компоненты такие свойства действия, как **Caption**, **Checked**, **Enabled**, **HelpContext**, **Hint**, **ImageIndex**, **ShortCut**, **Visible**. Впрочем, в любом компоненте разработчик может изменить переданное в него свойство. Обратной связи **TActionLink** с компонентами нет, так что эти изменения будут локальными и не отразятся на других компонентах. Если же требуется изменить свойства всех связанных с одним действием компонентов, надо изменять свойство объекта действия. Это облегчает программное управление компонентами, связанными с одним и тем же действием. Например, если в какой-то момент вам надо сделать недоступными (или, наоборот, доступными) два компонента — кнопку **Button3** и раздел меню **N5**, связанные с одним событием (назовем его объект **Do**), то при отсутствии централизованной диспетчеризации событий через **ActionList** вам пришлось бы писать два оператора:

```
Button3->Enabled = false;  
N5->Enabled = false;
```

а при наличии объекта **Do** — всего один:

```
Do->Enabled = false;
```

Дело не только в экономии кода, но и в его прозрачности, понятности его как для вас самих, так и для тех, кому придется, может быть, в дальнейшем сопровождать ваше приложение.

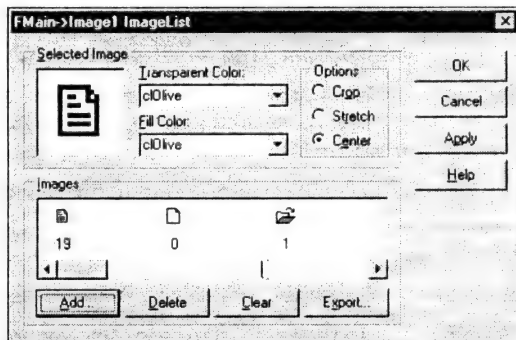
3.9.2 Список изображений — компонент **ImageList**

Компонент **ImageList** представляет собой набор изображений одинаковых размеров, на которые можно ссылаться по индексам, начинающимся с 0. Во многих рассмотренных ранее компонентах (меню, списках и др.) встречались свойства, представляющие собой ссылки на компонент **ImageList**. Этот компонент позволяет организовать эффективное и экономное управления множеством пиктограмм и битовых матриц. Он может включать в себя монохромные битовые матрицы, содержащие маски для отображения прозрачности рисуемых изображений.

Изображения в компонент **TImageList** могут быть загружены в процессе проектирования с помощью редактора списков изображений. Окно редактора, представленное на рис. 3.60, вызывается двойным щелчком на компоненте **TImageList** или щелчком правой кнопки мыши и выбором команды контекстного меню **ImageList Editor**.

Рис. 3.60

Окно редактора списков изображений



В окне редактора списков изображений вы можете добавить в списки изображения, пользуясь кнопкой **Add**, удалить изображение из списка кнопкой **Delete**, очистить весь список кнопкой **Clear**. При добавлении изображения в список открывается обычное окно открытия файлов изображений, в котором вы можете выбрать интересующий вас файл. Только учтите, что размер всех изображений в списке должен быть одинаковым. Как правило, это размер, используемый для пиктограмм в меню, списках, кнопках. При добавлении в список изображений для кнопок надо иметь в виду, что они часто содержат не одно, а два и более изображений (см. раздел 3.5.1). В этих случаях при попытке добавить изображение задается вопрос: «*Bitmap dimensions for ... are greater then imagelist dimensions. Separate into ... separate bitmaps?*» (Размерность изображения ... больше размерности списка. Разделить на ... отдельных битовых матрицы?). Если вы ответите отрицательно, то все изображения уменьшатся в горизонтальном размере и лягут как одно изображение. Использовать его в дальнейшем будет невозможно. Поэтому на заданный вопрос надо отвечать положительно. Тогда загружаемая битовая матрица автоматически разделится на отдельные изображения и потом вы можете удалить те из них, которые вам не нужны, кнопкой **Delete**.

Как видно из рис. 3.60, каждое загруженное в список изображение получает индекс. Именно на эти индексы впоследствии вы можете ссылаться в соответствующих свойствах разделов меню, списков, кнопок и т.д., когда вам надо загрузить в них то или иное изображение. Изменить последовательность изображений в списке вы можете просто перетащив изображение мышью на новое место.

В редакторе списков изображений вы можете, выделив то или иное изображение, установить его свойства: **Transparent Color** и **Fill Color**. Но это можно делать только в том сеансе работы с редактором списков изображений, в котором загружено данное изображение. Для изображений, загруженных в предыдущих сеансах, изменение этих свойств невозможно.

Свойство **Transparent Color** определяет цвет, который используется в маске для прозрачного рисования изображения. По умолчанию это цвет левого нижнего пикселя изображения. Для пиктограмм данное свойство устанавливается в **clNone**, поскольку пиктограммы уже маскированы.

Свойство **Fill Color** определяет цвет, используемый для заполнения пустого пространства при перемещении и центрировании изображения. Для пиктограмм данное свойство устанавливается в **clNone**.

Группа радиокнопок `Options` определяет способ размещения изображения битовой матрицы с размерами, не соответствующими размерам, принятым в списке:

Crop	Отображается часть изображения, помещающаяся в размер списка, начиная с левого верхнего угла
Stretch	Размеры изображения изменяются, становясь равными размерам списка. При этом возможны искажения
Center	Изображение центрируется, а если его размер больше размера списка, то не помещающиеся области отсекаются

Теперь рассмотрим основные свойства `TImageList`:

Свойство	Тип	Описание
Height	Integer	Высота изображений в списке
Width	Integer	Ширина изображений в списке
AllocBy	Integer	Определяет количество изображений, на которое увеличивается список для добавления новых изображений
Count	Integer	Определяет число изображений в списке. Свойство только для чтения

Остальные свойства определяют цвета и способы рисования изображений.

3.9.3 Приложение — компонент `ApplicationEvents` и объект `Application`

В каждом приложении автоматически создается объект `Application` типа `TApplication` — приложение. Этот компонент отсутствует в палитре библиотеки, вероятно, только потому, что он всегда один в приложении. `Application` имеет ряд свойств, методов, событий, характеризующих приложение в целом.

Рассмотрим сначала некоторые свойства `Application`. Булево свойство `Active` (только для чтения) характеризует активность приложения. Оно равно `true`, если форма приложения находится в фокусе. Если же пользователь переключился на работу с другим приложением, свойство `Active` равно `false`.

Свойство `ExeName` является строкой, содержащей имя выполняемого файла с полным путем к нему. Это свойство удобно использовать, чтобы определить каталог, из которого запущено приложение и который может содержать другие файлы (настройки, документы, базы данных и т.п.), связанные с приложением. Выражение `ExtractFilePath(Application->ExeName)` дает этот каталог. Обычно свойство `ExeName` тождественно функции `ParamStr(0)`, возвращающей нулевой параметр командной строки — имя файла с путем.

Свойство `Title` определяет строку, которая появляется около пиктограммы свернутого приложения. Если это свойство не изменяется во время выполнения, то оно равно опции `Title`, задаваемой во время проектирования на странице `Application` окна опций проекта (команда `Project | Options`). Свойство может изменяться программно, например, изменяя надпись в зависимости от режима работы приложения.

Свойство `MainForm` типа `TForm` определяет главную форму приложения. Булево свойство `ShowMainForm` определяет, должна ли главная форма быть видимой в момент запуска приложения на выполнение. По умолчанию оно равно `true`, что обеспечивает видимость главной формы в момент начала работы приложения.

Если же установить в головном файле проекта **Application->ShowMainForm** равным **false** до вызова метода **Application->Run()** и если при этом свойство **Visible** главной формы тоже равно **false**, то главная форма в первый момент будет невидимой.

Свойство **HelpFile** указывает файл справки, который используется в приложении в данный момент как файл по умолчанию. Если это свойство не изменяется во время выполнения, то оно равно опции **Help File**, задаваемой во время проектирования на странице **Application** окна опций проекта (команда **Project | Options**). Свойство можно изменять программно, назначая в зависимости от режима работы приложения тот или иной файл справки.

Ряд свойств объекта **Application** определяет ярлычки подсказок компонентов приложения. Свойство **Hint** содержит текст подсказки **Hint** того визуального компонента или раздела меню, над которым в данный момент перемещается курсор мыши. Смена этого свойства происходит в момент события **OnHint**, которое будет рассмотрено позднее. Во время этого события текст подсказки переносится из свойства **Hint** компонента, на который переместился курсор мыши, в свойство **Hint** объекта **Application**. Свойство **Application->Hint** можно использовать для отображения этой подсказки или для установки и отображения в полосе состояния текста, характеризующего текущий режим приложения.

Свойство **HintColor** типа **TColor** определяет цвет фона окна ярлычка. По умолчанию это цвет **clInfoBk**, но его значение можно изменять программно. Свойство **HintPause** определяет задержку появления ярлычка в миллисекундах после переноса курсора мыши на очередной компонент (по умолчанию 500 миллисекунд или половина секунды). Свойство **HintHidePause** аналогичным образом определяет интервал времени, после которого ярлычок становится невидимым (по умолчанию 2500 миллисекунд или две с половиной секунды). Свойство **HintShortPause** определяет аналогичным образом задержку перед появлением нового ярлычка, если в данный момент отображается другой ярлычок (по умолчанию 50 миллисекунд). Это свойство позволяет предотвратить неприятное мерцание, если пользователь быстро перемещает курсор мыши над разными компонентами.

Теперь остановимся на некоторых методах объекта **Application**. Методы **Initialize** — инициализация проекта, и **Run** — запуск выполнения приложения, включаются в каждый проект автоматически — вы можете это увидеть в головном файле проекта, если выполните команду **Project | View Source**. Там же вы можете увидеть применение метода создания форм **CreateForm** для всех автоматически создаваемых форм проекта. Если же в вашем проекте есть форма, которая исключена из списка автоматически создаваемых (команда **Project | Options** и соответствующая установка на странице **Forms**), то когда эта форма вам потребуется, вы должны будете вызвать этот метод:

```
void __fastcall CreateForm(System::TMetaClass* InstanceClass,
                           void *Reference);
```

где **InstanceClass** — класс создаваемой формы, который указывается операцией **__classid**, а **Reference** — ссылка на создаваемый объект (его имя). Например:

```
Application->CreateForm(__classid(TForm2), &Form2);
```

Метод **Terminate** завершает выполнение приложения. Если вам надо завершить приложение из главной формы, то вместо метода **Application->Terminate()** вы можете использовать метод **Close** главной формы. Но если вам надо закрыть приложение из какой-то вторичной формы, например, из диалога, то надо применять метод **Application->Terminate()**.

Метод **Minimize** сворачивает приложение, помещая его пиктограмму в полосу задач **Windows**.

Ряд методов связан с работой со справочными файлами. Выше уже говорилось о свойстве **HelpFile**, указывающем текущий файл справки. Метод **HelpContext**:

```
bool __fastcall HelpContext(Classes::THelpContext Context);
```

вызывает переход в файл справки на тему с идентификатором **Context**. Это идентификатор, который при протектировании справки поставлен в соответствие некоторой теме. Метод **HelpJump**:

```
bool __fastcall HelpJump(const System::AnsiString JumpID);
```

выполняет аналогичные действия, но его параметр **JumpID** — не идентификатор темы, а имя соответствующей темы в файле справки, задаваемое в нем сноской #.

Метод HelpCommand:

```
bool __fastcall HelpCommand(int Command, int Data);
```

позволяет выполнить указанную параметром **Command** команду API WinHelp с параметром **Data**. Метод генерирует событие **OnHelp** активной формы или приложения, а затем выполняет указанную команду WinHelp. Полный список команд WinHelp вы можете найти в теме WinHelp справочного файла **win32.hlp**, расположенного в каталоге ...\\Program Files\\Common Files\\Borland Shared\\MSHelp. Приведем только некоторые из них. Команда **HELP_CONTENTS** с параметром 0 отображает окно Содержание справки. Команда **HELP_INDEX** с параметром 0 отображает окно Указатель справки. Команда **HELP_CONTEXT** с параметром, равным идентификатору темы, отображает тему с заданным идентификатором (это тождественно рассмотренному ранее методу **HelpContext**). Команда **HELP_CONTEXTPOPUP** с параметром, равным идентификатору темы, делает то же самое, но отображает тему во всплывающем окне.

В классе **TApplication** имеется еще немало методов, но часть из них используется в явном виде очень редко (вы можете посмотреть их во встроенной справке C++Builder), а часть будет рассмотрена ниже при обсуждении событий объекта **Application**. Хотелось бы только обратить внимание читателя на очень полезный метод **MessageBox**, позволяющий вызывать диалоговое окно с указанным текстом, указанным заголовком и русскими надписями на кнопках (в русифицированных версиях Windows). Это наиболее удачный полностью русифицируемый стандартный диалог. См. о нем подробнее в главе 15 в разделе 15.7.2.3.

В классе **TApplication** определено множество событий, которые очень полезны для организации приложения. Ранее для использования этих событий было необходимо вводить соответствующие обработчики и указывать на них объекту **Application** специальными операторами. В C++Builder 5 введен компонент **ApplicationEvents**, существенно облегчивший эту задачу. Этот компонент перехватывает события объекта **Application** и, следовательно, обработчики этих событий теперь можно писать как обработчики событий невизуального компонента **ApplicationEvents**. На каждой форме приложения можно разместить свой компонент **ApplicationEvents**. События объекта **Application** будут передаваться всем этим компонентам. Если вы хотите, чтобы событие передавалось прежде всего какому-то одному из них, примените к нему метод **Activate**, который поставит его в начало очереди компонентов **ApplicationEvents**. Если же вы при этом не хотите, чтобы другие компоненты **ApplicationEvents** получали события, примените к привилегированному компоненту метод **CancelDispatch**. Тогда после обработки события в данном компоненте другие компоненты **ApplicationEvents** вообще не будут реагировать на эти события.

Во многие обработчики событий компонента **ApplicationEvents** передается по ссылке параметр **Handled**. По умолчанию его значение равно **false**. Если вы обработали соответствующее событие и не хотите, чтобы оно далее обрабатывалось другими компонентами **ApplicationEvents**, надо в обработчике установить **Handled = true**. Если же вы оставите **Handled = false**, то событие будут пытаться обрабатывать другие компоненты **ApplicationEvents** (если они есть). Если событие так и останется необработанным, то его будет пытаться обработать активный компонент, а

если не обработает — то активная форма. Предотвратить обработку события другими компонентами можно, используя описанный ранее метод **CancelDispatch**.

Ниже приведена таблица событий компонента **ApplicationEvents** с их краткими описаниями.

Событие	Описание
OnActionExecute	Возникает при выполнении (Execute) некоторого действия, объявленного в компоненте ActionList , но не обработанного им (не написан соответствующий обработчик). Инициализация этого события может быть, например, выполнена методом Application->ExecuteAction(<имя действия>) . Если событие не обработано в ActionList , то оно может быть обработано на уровне приложения. В обработчик OnActionExecute передается параметр Action — действие и по ссылке передается параметр Handled (см. выше).
OnActionUpdate	Возникает при обновлении (Update) некоторого действия, объявленного в компоненте ActionList , но не обработанного им (не написан соответствующий обработчик). Если событие не обработано в ActionList , то оно может быть обработано на уровне приложения. В обработчик OnActionUpdate передается параметр Action — действие и по ссылке передается параметр Handled (см. выше).
OnActivate	Возникает, когда приложение становится активным. Это происходит при начале выполнения и в случаях, когда пользователь, перейдя к другим приложениям, вернулся в данное. Если это событие обработано в ApplicationEvents , то предотвратить дальнейшую его обработку можно методом CancelDispatch .
OnDeactivate	Возникает перед тем моментом, когда приложение перестает быть активным (пользователь переключается на другое приложение). Если событие обработано в ApplicationEvents , то предотвратить дальнейшую его обработку можно методом CancelDispatch .
OnException	Возникает, когда в приложении сгенерировано исключение, которое нигде не перехвачено. В обработчик передается параметр Sender — источник исключения, и параметр E типа Exception — объект исключения. Параметр E помогает определить тип исключения. Например, if (E->ClassNameIs("EZeroDivide")) В обработчике события OnException вы можете предусмотреть нестандартную обработку исключений на уровне приложения, например, русифицировать стандартные сообщения об исключениях и дать пользователю какие-то рекомендации. Учтите, что введение вами обработчика OnException отключит стандартную реакцию приложения на исключительные ситуации.

Событие	Описание
OnHelp	Возникает при запросе приложением справки. Это событие возникает, в частности, при выполнении рассмотренных ранее методов приложения HelpContext , HelpJump и HelpCommand . Обработчик может использоваться для каких-то подготовительных операций, например, для задания файла справки (параметр Application->HelpFile). В обработчик передаются параметры — Command команда API WinHelp (см. выше описание HelpCommand), Data — параметр этой команды и по ссылке передается булев параметр CallHelp . Если его установить в true , то после завершения обработчика будет вызван WinHelp, в противном случае вызова WinHelp не будет.
OnHint	Возникает в момент, когда курсор мыши начинает перемещаться над компонентом или разделом меню, в котором определено свойство Hint . При этом свойство Hint компонента переносится в свойство Hint приложения (Application->Hint) и в обработчике данного события может отображаться, например, в строке состояния.
OnIdle	Возникает, когда приложение начинает простаивать, ожидая, например, действий пользователя. В обработчик передается по ссылке булев параметр Done , который по умолчанию равен true . Если оставить его без изменения, то по окончании работы обработчика данного события будет вызвана функция WaitMessage API Windows, которая займется в период ожидания другими приложениями. Если задать Done = false , то эта функция вызываться не будет. Это может снизить производительность Windows.
OnMessage	Возникает, когда приложение получает сообщение Windows (но не переданное функцией SendMessage API Windows). В обработчике события OnMessage можно предусмотреть нестандартную (отличную от определенной в TApplication) обработку сообщения. В обработчик передается параметр Msg — полученное сообщение и по ссылке передается параметр Handled (см. выше). Учтите, что в Windows передаются тысячи сообщений в секунду, так что обработчик OnMessage может серьезно снизить производительность приложения.
OnMinimize	Возникает при сворачивании приложения.
OnRestore	Возникает при восстановлении ранее свернутого приложения.
OnShortCut	Возникает при нажатии пользователем клавиши. Событие возникает до того, как возникло стандартное событие OnKeyDown компонента или формы. Обработчик позволяет предусмотреть нестандартную реакцию на нажатие какой-то клавиши. В него передается параметр сообщения Windows Msg , поле CharCode которого (Msg.CharCode) содержит виртуальный код нажатой клавиши. Передается также по ссылке параметр Handled . Если задать ему значение true , то стандартные события OnKeyDown , OnKeyPress , OnKeyUp не наступят.

Событие	Описание
OnShowHint	Возникает, когда приложение собирается отобразить ярлычок с текстом подсказки Hint . В обработчик передается по ссылке параметр HintStr — первая часть свойства Hint компонента, ярлычок которого должен отображаться. В обработчике этот текст можно изменить. Так же по ссылке передается параметр CanShow . Если в обработчике установить его равным false , то ярлычок отображаться не будет. Третий параметр, передаваемый по ссылке — HintInfo . Это структура, поля которой (см. встроенную справку C++Builder) содержат информацию о ярлычке: его координаты, цвет, задержки появления и т.п. В частности, имеется поле HintControl — компонент, сообщение которого должно отображаться в ярлычке, и поле HintStr — отображаемое сообщение. По умолчанию HintInfo.HintStr — первая часть свойства Hint компонента. Но в обработчике это значение можно изменить.

Приведем примеры использования событий компонента **ApplicationEvents**. Обработчик события **OnHint**:

```
void __fastcall TForm1::ApplicationEvents1Hint(TObject *Sender)
{
    StatusBar1->SimpleText = Application->Hint;
}
```

отображает в полосе состояния **StatusBar1** (см. раздел 3.7.7) вторую часть свойства **Hint** любого компонента, в котором определено это свойство и над которым перемещается курсор мыши. Отображение происходит независимо от значения свойства **ShowHint** компонента.

Обработчик события **OnShowHint**:

```
void __fastcall TForm1::ApplicationEvents1ShowHint(
    AnsiString &HintStr, bool &CanShow,
    THintInfo &HintInfo)
{
    if (HintInfo.HintControl->ClassNameIs("TEdit"))
        if (Canvas->TextWidth(Edit1->Text) > Edit1->ClientWidth)
        {
            HintStr = Edit1->Text;
            ApplicationEvents1->CancelDispatch();
        }
}
```

проверяет класс источника события и, если это окно редактирования, то с помощью функции **TextWidth** проверяет, не превышает ли длина текста ширины клиентской области. Если превышает, то текст ярлычка подменяется текстом, содержащимся в окне редактирования. Такой прием позволяет пользователю, подведя курсор мыши к окну редактирования, увидеть во всплывающем окне полный текст, который может не быть виден обычным образом, если он длинный и не помещается целиком в окне редактирования. Только учтите, что событие **OnShowHint** будет наступать при перемещении курсора только над теми компонентами, в которых свойство **ShowHint** установлено в **true**. Если вы хотите запретить отображение в ярлычке собственной подсказки **Hint** в окнах редактирования, содержащих короткий текст, то приведенный код надо дополнить оператором:

```
else CanShow = false;
```

Обработчик события **OnHelp**:

```
bool __fastcall TForm1::ApplicationEvents1Help(
    WORD Command, int Data, bool &CallHelp)
```

```

{
    if ((Command == HELP_CONTEXT) && (Data < 10))
    {
        Application->HelpCommand(HELP_CONTEXTPOPUP, Data);
        CallHelp = false;
    }
    return true;
}

```

обеспечивает отображение всех контекстных справок с номерами идентификаторов тем, меньшими 10, во всплывающем окне, не вызывая при этом WinHelp. Это дает возможность отобразить многострочные (в отличие от ярлычков) всплывающие окна, поясняющие назначение тех или иных элементов приложения. Событие **OnHelp** наступает, если пользователь нажал клавишу F1 в компоненте, для которого задано значение свойства **HelpContext** — идентификатор справки, или если в заголовке формы имеется системная кнопка справки и с ее помощью пользователь сделал запрос о назначении компонента, для которого задано значение **HelpContext**. В последнем случае обработчик события может проанализировать состояние приложения и в зависимости от режима работы изменить идентификатор темы справки.

Обработчик события **OnShortCut**:

```

void __fastcall TForm1::ApplicationEvents1ShortCut(
                                                    TWMKey &Msg, bool &Handled)
{
    if (Msg.CharCode == 'Q')
    if (Application->MessageBox(
        "Действительно хотите завершить работу?",
        "Подтвердите завершение",
        MB_YESNOCANCEL+MB_ICONQUESTION) == IDYES)
        Application->Terminate();
}

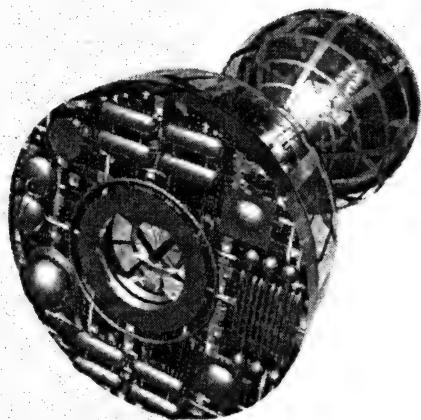
```

перехватывает нажатие пользователем клавиш и, если нажата клавиша с символом «Q» (в любом регистре и независимо от установки русского или английского языка), то пользователю методом **Application->MessageBox** предлагается диалоговое окно с запросом о завершении работы. Если пользователь подтверждает завершение, то приложение закрывается методом **Application->Terminate**.

Часть II

Разработка приложений для Windows

- Глава 4 Проектирование графического интерфейса пользователя
- Глава 5 Графика и мультимедиа
- Глава 6 Взаимодействие приложения с внешними программами
- Глава 7 Повторное использование разработанных кодов
- Глава 8 Разработка справочной системы (создание файлов .hlp)



Проектирование графического интерфейса пользователя

4.1 Требования к интерфейсу пользователя приложений для Windows

4.1.1 Общие рекомендации по разработке графического интерфейса

Под графическим интерфейсом пользователя (Graphical User Interface — GUI) подразумевается тип экранного представления, при котором пользователь может выбирать команды, запускать задачи и просматривать списки файлов, указывая на пиктограммы или пункты в списках меню, показанных на экране. Действия могут, как правило, выполняться с помощью мыши, либо нажатием клавиш на клавиатуре. Типичным примером графического интерфейса пользователя является Windows.

C++Builder предоставляет разработчику приложения широкие возможности быстрого и качественного проектирования графического интерфейса пользователя — различных окон, кнопок, меню и т.д. Так что разработчик может в полной мере проявить свою фантазию. Но полеты фантазии очень полезно ограничивать. Есть определенные принципы построения графического интерфейса пользователя, и пренебрегающий ими обречен на то, что его приложение будет выглядеть чужеродным объектом в среде Windows.

Для пользователя одним из принципиальных преимуществ работы с Windows является то, что большинство имеющихся приложений выглядят и ведут себя сходным образом. После того, как вы поработаете с несколькими приложениями, вы обнаружите, что можете заранее почти наверняка сказать, где можно найти ту или иную функцию в программе, которую только что приобрели, или какие быстрые клавиши надо использовать для выполнения тех или иных операций.

Хороший стиль программирования

Ваше приложение не должно выбиваться из общего стиля Windows ни своим внешним видом (шрифтами, цветом, меню), ни организацией диалогов. Фантазию надо обращать на построение эффективных и надежных алгоритмов функционирования, а не на экзотическое оформление приложения.

Фирма Microsoft предложила спецификации для разработки программного обеспечения Windows, направленные на то, чтобы пользователь не тратил время на освоение нюансов пользовательского интерфейса новой программы, чтобы он смог как можно скорее продуктивно применять ваше приложение. Эти спецификации образуют основу программы логотипа Windows, проводящейся Microsoft. Чтобы вы могли поставить на свой программный продукт штамп «Разработано для Windows», ваша программа должна удовлетворять определенным критериям.

Конечно, вряд ли вы будете очень озабочены приобретением официального права на логотип Windows, если только не разработали сногшибательную программу широкого применения, которую надеетесь успешно продавать на международном рынке. Но прислушаться к рекомендациям по разработке графического

интерфейса пользователя в любом случае полезно. Они основаны на психофизиологических особенностях человека и существенно облегчат жизнь будущим пользователям вашей программы, увеличат производительность их работы.

4.1.2 Многооконные приложения

Чаще всего сколько-нибудь сложное приложение не может ограничиться одним окном. Поэтому прежде всего вам нужно решить вопрос управления окнами. Есть две различные модели приложений: с интерфейсом одного документа (SDI) и с интерфейсом множества документов (MDI).

В большинстве случаев следует отдавать предпочтение интерфейсу SDI. Этот интерфейс не обязательно предполагает наличие действительно только одного окна, как в приложениях Windows, типа «Калькулятор». Такое приложение, как «Проводник» Windows, также является SDI приложением, но в нужные моменты оно создает вторичные окна для поиска файлов или папок, задания параметров, просмотра свойств файлов и других целей.

С другой стороны, у приложений MDI тоже есть свои преимущества. Хороший пример такого приложения — Microsoft Word. В приложении MDI имеется родительское (первичное) окно и ряд дочерних окон (называемых также *окнами документов*). Бывают ситуации, когда выгодно отображать информацию в нескольких окнах, которые совместно используют элементы интерфейса (например, меню или инструментальные линейки). Окна документов управляются и ограничиваются родительским окном. Если вы уменьшаете размер родительского окна, то дочерние окна могут исчезать из поля зрения.

Случаи, когда нужно использовать модель MDI, довольно редки. Прежде всего, это следует делать только тогда, когда все дочерние окна будут содержать идентичные объекты — например, текстовые документы или электронные таблицы. Не применяйте MDI, если вы собираетесь работать в приложении с дочерними окнами разного типа (например, текстовыми документами и электронными таблицами одновременно). Не применяйте MDI, если вы хотите управлять тем, какое из дочерних окон должно находиться поверх других, используя свойство «всегда наверху», или если вы хотите управлять размерами окон, делать их невидимыми и т. п. Интерфейс MDI предназначен для очень узкого диапазона приложений, в которых все дочерние окна однородны (как это имеет место в Word или Excel). Приспособить его к чему-то другому не получится. Наконец, следует заметить, что Microsoft не поощряет разработку новых приложений MDI (в основном потому, что для Windows было написано слишком много плохих программ этого типа). Подробнее о технологии построения приложений MDI будет рассказано в разделе 4.5.4.

4.1.3 Стиль окон приложения

Основным элементом любого приложения является форма — контейнер, в котором размещаются другие визуальные и не визуальные компоненты. С точки зрения пользователя форма — это окно, в котором он работает с приложением. К внешнему виду окон в Windows предъявляются определенные требования. К счастью, C++Builder автоматически обеспечивает стандартный для Windows вид окон вашего приложения. Но вам надо продумать и указать, какие кнопки в полосе системного меню должны быть доступны в том или ином окне, должно ли окно допускать изменение пользователем его размеров, каким должен быть заголовок окна. Все эти характеристики окон обеспечиваются установкой и управлением свойствами формы.

Свойство **BorderStyle** определяет общий вид окна и операции с ним, которые разрешается выполнять пользователю. Это свойство может принимать следующие значения:

bsSizeable	Обычный вид окна Windows с полосой заголовка, с возможностью для пользователя изменять размеры окна с помощью кнопок в полосе заголовка или с помощью мыши, потянув за какой-либо край окна. Это значение BorderStyle задается по умолчанию
bsDialog	Неизменяемое по размерам окно. Типичное окно диалогов
bsSingle	Окно, размер которого пользователь не может изменить, потянув курсором мыши край окна, но может менять кнопками в полосе заголовка
bsToolWindow	То же, что bsSingle , но с полосой заголовка меньшего размера
bsSizeToolWin	То же, что bsSizeable , но с полосой заголовка меньшего размера и с отсутствием в ней кнопок изменения размера
bsNone	Без полосы заголовка. Окно не только не допускает изменения размера, но и не позволяет переместить его по экрану

Свойство **BorderIcons** определяет набор кнопок, которые имеются в полосе заголовка. Множество кнопок задается элементами:

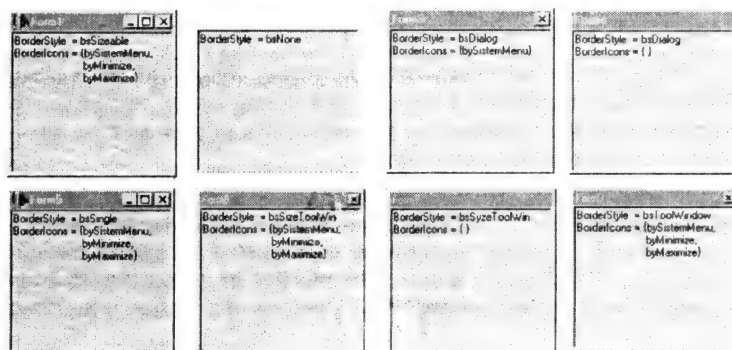
bySistemMenu	кнопка системного меню (для Windows 95/98/2000 и NT это кнопка с крестиком, закрывающая окно)
byMinimize	кнопка Свернуть, сворачивает окно до пиктограммы
byMaximize	кнопка Развернуть, разворачивает окно на весь экран
byHelp	кнопка справки

Следует отметить, что не все кнопки могут появляться при любых значениях **BorderStyle**.

На рис. 4.1 представлен вид окон форм во время выполнения при некоторых сочетаниях свойств **BorderStyle** и **BorderIcons**. Для создания диалоговых окон обычно используется стиль заголовка **bsDialog** (формы **Form3** и **Form4** на рис. 4.1), причем в этих окнах можно исключить кнопку системного меню (форма **Form4**) и в этом случае пользователь не может закрыть окно никакими способами, кроме как выполнить какие-то предписанные ему действия на этой форме. При стиле **bsNone** пользователь не может изменить ни размер, ни положение окна на экране. Формы **Form3**, **Form6** и **Form8** внешне различаются только размером полосы заголовка, уменьшенным в двух последних формах. Но между ними есть и принципиальное различие: размер формы **Form6** (стиль заголовка **bsSizeToolWin**) пользова-

Рис. 4.1

Формы при разных сочетаниях свойств **BorderStyle** и **BorderIcons**



тель может изменить, потянув курсором мыши за край окна. Для форм **Form3** и **Form8** это невозможно. Обратите также внимание, что в формах **Form6** и **Form8** задание кнопок свертывания и разворачивания окна никак не влияет на его вид: эти кнопки просто не могут появляться в этих стилях полос заголовков окон.

Хороший стиль программирования

Без особой необходимости не делайте окна приложения с изменяемыми пользователем размерами. При изменении размеров, если не применены специальные приемы, описанные в разделе 4.2, нарушается компоновка окна и пользователь ничего не выигрывает от своих операций с окном. Окно имеет смысл делать с изменяемыми размерами, только если это позволяет пользователю изменять полезную площадь каких-то расположенных в нем компонентов отображения и редактирования информации: текстов, изображений, списков и т.п.

Хороший стиль программирования

Для основного окна приложения с неизменяемыми размерами наиболее подходящий стиль — `BorderStyle = bsSingle` с исключением из числа доступных кнопок кнопки Развернуть (`BorderIcons.byMaximize = false`). Это позволит пользователю сворачивать окно, восстанавливать, но не даст возможности развернуть окно на весь экран или изменить размер окна.

Хороший стиль программирования

Для вторичных диалоговых окон наиболее подходящий стиль — `BorderStyle = bsDialog`. Можно также использовать `BorderStyle = bsSingle`, одновременно исключая из числа доступных кнопок кнопку Развернуть (задавая `BorderIcons.byMaximize = false`). Это позволит пользователю сворачивать диалоговое окно, если оно заслоняет на экране что-то нужное ему, восстанавливать окно, но не даст возможности развернуть окно на весь экран или изменить размер окна.

Предупреждение

Избегайте, как правило, стиля `BorderStyle = bsNone`. Невозможность переместить окно может создать пользователю трудности, если окно заслонит на экране что-то интересное пользователю.

Свойство формы **WindowState** определяет вид, в котором окно первоначально предьявляется пользователю при выполнении приложения. Оно может принимать значения:

wsNormal	нормальный вид окна (это значение WindowState используется по умолчанию)
wsMinimized	окно свернуто
wsMaximized	окно развернуто на весь экран

Если свойство **WindowState** имеет значение **wsNormal** или пользователь, манипулируя кнопками в полосе заголовка окна, привел окно в это состояние, то положение окна при запуске приложения определяется свойством **Position**, которое может принимать значения:

poDesigned	Первоначальные размеры и положение окна во время выполнения те же, что во время проектирования. Это значение принимается по умолчанию, но обычно его следует изменить
-------------------	---

poScreenCenter	Окно располагается в центре экрана. Размер окна тот, который был спроектирован. В мультиэкранных приложениях (см. раздел 4.5.6), работающих одновременно с множеством мониторов эта центральная позиция может быть несколько изменена, чтобы изображение попало точно на один монитор, определяемый свойством DefaultMonitor
poDesktopCenter	Окно располагается в центре экрана. Размер окна тот, который был спроектирован. Этот режим не приспособивается к приложениям с множеством мониторов (см. раздел 4.5.6)
poDefault	Местоположение и размер окна определяет Windows, учитывая размер и разрешение экрана. При последовательных показах окна его положение сдвигается немного вниз и вправо
poDefaultPosOnly	Местоположение окна определяет Windows. При последовательных показах окна его положение сдвигается немного вниз и вправо. Размер окна — спроектированный
poDefaultSizeOnly	Размер окна определяет Windows, учитывая размер и разрешение экрана. Положение окна — спроектированное
poMainFormCenter	Это значение предусмотрено только начиная с C++Builder 5. Окно располагается в центре главной формы. Размер окна тот, который был спроектирован. Этот режим не приспособивается к приложениям с множеством мониторов (см. раздел 4.5.6). Используется только для вторичных форм. Для главной формы действует так же, как poScreenCenter

Хороший стиль программирования

Обычно целесообразно для главной формы приложения задавать значение **Position** равным **poScreenCenter** или **poDefaultPosOnly**. И только в сравнительно редких случаях, когда на экране при выполнении приложения должно определенным образом располагаться несколько окон, имеет смысл оставлять значение **poDesigned**, принимаемое по умолчанию.

Если выбранное значение свойства **Position** предусматривает выбор размера формы самим Windows по умолчанию, то на этот выбор влияют свойства **PixelsPerInch** и **Scaled**. По умолчанию первое из них задается равным количеству пикселей на дюйм в системе, второе установлено в **false**. Если задать другое число пикселей на дюйм, то свойство **Scaled** автоматически становится равным **true**. В этом случае при запуске приложения размер формы будет изменяться в соответствии с пересчетом заданного числа пикселей на дюйм к реальному числу пикселей на дюйм в системе (но только при разрешающем это значении свойства **Position**).

Свойство **AutoScroll** определяет, будут ли на форме в процессе выполнения появляться автоматически полосы прокрутки в случае, если при выбранном пользователем размере окна не все компоненты помещаются в нем. Если значение **AutoScroll** равно **true**, то будут. В противном случае при уменьшении размера окна пользователь теряет доступ к компонентам, не поместившимся в его поле.

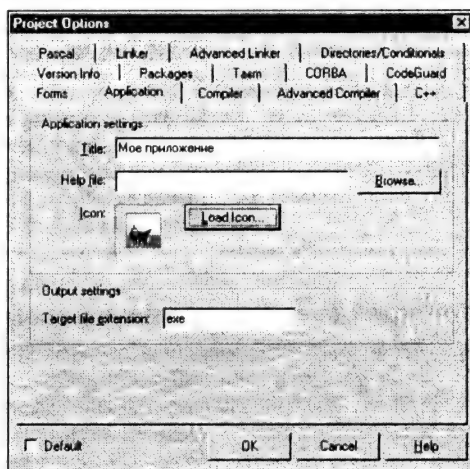
Свойство **Icon** задает пиктограмму формы. По умолчанию используется стандартная пиктограмма C++Builder. Нажав в Инспекторе Объектов кнопку с тремя точками в строке свойства **Icon**, вы попадаете в окно Редактора Изображений (Picture Editor) (см. рис. 3.29 в разделе 3.5.1 или рис. 5.1 в разделе 5.1.1.1). Щелкнув в нем на кнопке Load (загрузить), вы можете выбрать любой файл с изображе-

нием пиктограммы (файл с расширением **.ico**). С C++Builder поставляется некоторое число пиктограмм, расположенных в каталоге `Images\icons`.

Свойство **Icon** задает только пиктограмму формы, которая отображается в левом верхнем углу окна приложения в его нормальном состоянии. Но если пользователь свернет окно, то в полосе задач будет видна другая пиктограмма — пиктограмма приложения. Ту же пиктограмму увидит пользователь, если будет просматривать средствами Windows содержимое каталога. По умолчанию для нее используется стандартная пиктограмма C++Builder. При сворачивании приложения рядом с пиктограммой в полосе задач пользователь будет видеть надпись — по умолчанию это имя приложения. Если вы хотите, то можете изменить эту пиктограмму и эту надпись. Для этого вы должны выполнить команду **Project | Options** и в открывшемся окне опций проекта перейти на страницу **Application** (рис. 4.2). В этом окне вы можете задать заголовок (**Title**), который увидит пользователь в полосе задач при сворачивании приложения. А кнопка **Load Icon** позволяет вам выбрать пиктограмму, которая будет видна в полосе задач при сворачивании приложения или при просмотре пользователем каталога, в котором расположен выполняемый файл приложения.

Рис. 4.2

Страница **Application** окна опций проекта



Одно из основных свойств формы — **FormStyle**, которое может принимать значения:

fsNormal	Окно обычного приложения. Это значение FormStyle принято по умолчанию
fsMDIForm	Родительская форма приложения MDI, т.е. приложения с дочерними окнами, используемого при работе с несколькими документами одновременно
fsMDIChild	Дочерняя форма приложения MDI
fsStayOnTop	Окно, остающееся всегда поверх остальных окон Windows

О приложениях MDI подробно будет рассказано в разделе 4.5.4. Так что значения **fsMDIForm** и **fsMDIChild** мы пока подробнее обсуждать не будем. А значение **FormStyle = fsStayOnTop** делает окно всегда остающимся на экране поверх остальных окон не только данного приложения, но и всех других приложений, в которые может перейти пользователь.

Хороший стиль программирования

Используйте стиль `FormStyle = fsStayOnTop` для отображения окон сообщений пользователю о каких-то аварийных ситуациях.

В ряде случаев полезно предоставить пользователю самому решать, сделать ли данное окно располагающимся всегда поверх остальных, или нет. Например, ему может временно потребоваться перейти в какое-то другое приложение, чтобы получить необходимую информацию, и при этом будет хотеться видеть поверх этого приложения ваше окно, чтобы сравнивать в этих двух окнах какие-то данные. Такую возможность легко предоставить пользователю. Введите в меню вашего окна раздел Поверх остальных и в обработчик щелчка на этом разделе вставьте операторы

```
MStayOnTop->Checked = ! MStayOnTop->Checked;  
if (MStayOnTop->Checked)  
    Form1->FormStyle = fsStayOnTop;  
else Form1->FormStyle = fsNormal;
```

В этом коде подразумевается, что объект раздела меню, о котором идет речь, назван **MStayOnTop**. Тогда при выборе пользователем этого раздела меню в нем появится индикатор (см. раздел 3.6.1), а окно приобретет статус расположенного всегда поверх остальных. При повторном выборе этого раздела индикатор исчезнет и окно приобретет обычный статус.

4.1.4 Цветовое решение приложения

Цвет является мощным средством воздействия на психику человека. Именно поэтому обращаться с ним надо очень осторожно. Неудачное цветовое решение может приводить к быстрому утомлению пользователя, работающего с вашим приложением, к рассеиванию его внимания, к частым ошибкам. Слишком яркий или неподходящий цвет может отвлекать внимание пользователя или вводить его в заблуждение, создавать трудности в работе. А удачно подобранная гамма цветов, осмысленные цветовые акценты снижают утомляемость, сосредоточивают внимание пользователя на выполняемых в данный момент операциях, повышают эффективность работы. С помощью цвета вы можете на что-то намекнуть или привлечь внимание к определенным областям экрана. Цвет может также связываться с различными состояниями объектов.

Надо стремиться использовать ограниченный набор цветов и уделять внимание их правильному сочетанию. Расположение ярких цветов, таких, как красный, на зеленом или черном фоне затрудняет возможность сфокусироваться на них. Не рекомендуется использовать дополнительные цвета. Обычно наиболее приемлемым цветом для фона будет нейтральный цвет, например, светло-серый (используется в большинстве продуктов Microsoft). Помните также, что яркие цвета кажутся выступающими из плоскости экрана, в то время как темные как бы отступают вглубь.

Цвет не должен использоваться в качестве основного средства передачи информации. Можно использовать различные панели, формы, штриховку и другие методики выделения областей экрана. Microsoft даже рекомендует разрабатывать приложение сначала в черно-белом варианте, а уже потом добавлять к нему цвет.

Нельзя также забывать, что восприятие цвета очень индивидуально. А по оценке Microsoft девять процентов взрослого населения вообще страдают нарушениями цветовосприятия. Поэтому не стоит навязывать пользователю свое видение цвета, даже если оно безукоризненно. Надо предоставить пользователю возможность самостоятельной настройки на наиболее приемлемую для него гамму. К тому же не стоит забывать, что может быть кто-то захочет использовать вашу программу на машине с монохромным монитором.

Посмотрим теперь, как задаются цвета приложения, разрабатываемого в C++Builder. Большинство компонентов имеют свойство **Color** (цвет), который вы можете изменять в Инспекторе Объектов при проектировании или программно во время выполнения (если хотите, чтобы цвета в различных режимах работы приложения были разные). Щелкнув на этом свойстве в Инспекторе Объектов, вы можете увидеть в выпадающем списке большой набор предопределенных констант, обозначающих цвета. Все их можно разбить на две группы: статические цвета типа **clBlack** — черный, **clGreen** — зеленый и т.д., и системные цвета типа **clWindow** — текущий цвет фона окон, **clMenuText** — текущий цвет текста меню и т.д. Полный список всех констант и их описание, а также способ конструирования любого другого статического цвета вы можете найти в справочной части книги в главе 16 в разделе «Color».

Статические цвета вы выбираете сами и они будут оставаться неизменными при работе приложения на любом компьютере. Это не очень хорошо, поскольку пользователь не сможет адаптировать вид вашего приложения к своим потребностям. При выборе желательной ему цветовой схемы пользователь может руководствоваться самыми разными соображениями: начиная с практических (например, он может хотеть установить черный фон, чтобы экономить энергию батареи), и кончая эстетическими (он может предпочитать, например, шкалу оттенков серого, потому что не различает цвета). Все это он не может делать, если вы задали в приложении статические цвета. Но уж если по каким-то соображениям вам надо их задать, старайтесь использовать базовый набор из 16 цветов. Если вы попытаетесь использовать 256 (или, что еще хуже, 16 миллионов) цветов, это может замедлить работу вашего приложения, или оно будет выглядеть плохо на машине пользователя с 16 цветами. К тому же подумайте (а, как правило, это надо проверить и экспериментально), как будет выглядеть ваше приложение на монохромном дисплее.

Исходя из изложенных соображений, везде, где это имеет смысл, следует использовать для своего приложения палитру системных цветов. Это те цвета, которые устанавливает пользователь при настройке Windows. Когда вы создаете новую форму или размещаете на ней компоненты, C++Builder автоматически присваивает им цвета в соответствии со схемой цветов, установленной в Windows. Конечно, вы будете менять эти установки по умолчанию. Но если при этом вы используете соответствующие константы системных цветов, то, когда пользователь изменит цветовую схему оформления экрана Windows, ваше приложение также будет соответственно меняться и не будет выпадать из общего стиля других приложений.

Хороший стиль программирования

Не злоупотребляйте в приложении яркими цветами. Пестрое приложение — обычно признак дилетантизма разработчика, утомляет пользователя, рассеивает его внимание. Как правило, используйте системные цвета, которые пользователь может перестраивать по своему усмотрению. Из статических цветов обычно имеет смысл использовать только **clBlack** — черный, **clWhite** — белый и **clRed** — красный цвет предупреждения об опасности.

Единству цветового решения отдельных частей экрана способствует также использование свойства **ParentColor**. Если это свойство установлено в **true**, то цвет компонента соответствует цвету содержащего его контейнера или формы. Это обеспечивает единство цветового решения окна и, кроме того, позволяет программно изменять цвет сразу группы компонентов, если вы, например, хотите, чтобы их цвет зависел от текущего режима работы приложения. Для такого группового изменения достаточно изменить только цвет контейнера.

4.1.5 Шрифты текстов

Шрифт надписей и текстов компонентов C++Builder задается свойством **Font**, имеющим множество подсвойств. Кроме того, в компонентах имеется свойство **ParentFont**. Если это свойство установлено в **true**, то шрифт данного компонента берется из свойства **Font** его родительского компонента — панели или формы, на которой расположен компонент. Использование свойств **ParentFont** и **ParentColor** помогает обеспечить единообразие отображения компонентов в окне приложения.

По умолчанию для всех компонентов C++Builder задается имя шрифта **MS Sans Serif** и размер — 8. Константа множества символов **Charset** задается равной **DEFAULT_CHARSET**. Последнее означает, что шрифт выбирается только по его имени и размеру. Если описанный шрифт недоступен в системе, то Windows заменит его другим шрифтом.

Чаще всего эти установки по умолчанию можно не изменять. Конечно, никто не мешает задать для каких-то компонентов другой размер шрифта или атрибуты типа полужирный, курсив и т.д. Но изменять имя шрифта для вашего приложения надо с определенной осторожностью. Дело в том, что шрифт, установленный на вашем компьютере, не обязательно должен иметься и на компьютере пользователя. Поэтому использование какого-то экзотического шрифта может привести к тому, что пользователь, запустив ваше приложение на своем компьютере, увидит вместо русского текста абракадабру на никому не понятном языке. Чтобы избежать таких казусов, вам придется прикладывать к своему приложению еще и файлы использованных шрифтов и пояснять пользователю, как он должен установить их на своем компьютере, если они там отсутствуют. Или вводить автоматическую проверку и установку нужных вам шрифтов в установочную программу вашего приложения.

Использование шрифтов по умолчанию: **System** или **MS Sans Serif**, чаще всего позволяет избежать подобных неприятностей. Впрочем, увы, не всегда. Если вы используете для надписей русские тексты, то при запуске приложения на компьютере с нерусифицированным Windows иногда возможны неприятности. Для подобных случаев все-таки полезно приложить файлы использованных шрифтов к вашей программе. Вы можете при установке вашего приложения узнать, имеется ли на компьютере пользователя нужный шрифт, например, с помощью следующего кода:

```
if (Screen->Fonts->IndexOf("Arial Cir") == -1)
    ...
```

В этом коде многоточием обозначены действия, которые надо выполнить, если нужного шрифта (в примере — **Arial Cir**) на компьютере нет. Эти действия могут заключаться в копировании файлов шрифта с установочной дискеты или CD ROM на компьютер пользователя.

Другой выход из положения — ввести в приложение команду выбора шрифта пользователем. Это позволит ему выбрать подходящий шрифт из имеющихся в его системе. Осуществляется подобный выбор с помощью стандартного диалога, оформленного в виде компонента **FontDialog** (см. раздел 3.8.4 главы 3). Проведенную пользователем установку можно запоминать в файле **.INI**, в реестре или в другом файле конфигурации и читать автоматически информацию из этого файла при каждом запуске приложения (см. раздел 4.7.2).

Подробное рассмотрение всех свойств компонентов, связанных со шрифтами, примеры их использования и исследования вы найдете в главе 16 в разделе «Font».

4.1.6 Меню

4.1.6.1 Требования к меню

Практически любое приложение должно иметь меню, поскольку именно меню дает наиболее удобный доступ к функциям программы. Существует несколько различных типов меню: главное меню с выпадающими списками разделов, каскадные меню, в которых разделу первичного меню ставится в соответствие список подразделов, и всплывающие или контекстные меню, появляющиеся, если пользователь щелкает правой кнопкой мыши на каком-то компоненте.

В C++Builder меню создаются компонентами **MainMenu** — главное меню, и **PopupMenu** — всплывающее меню. Оба компонента расположены на странице Standard. Все операции по проектированию меню с помощью этих компонентов подробно рассмотрены в разделе 3.6. А в данном разделе мы обсудим требования, предъявляемые к меню приложений Windows.

Основное требование к меню — их стандартизация. Это требование относится ко многим аспектам меню: месту размещения заголовков меню и их разделов, форме самих заголовков, клавишам быстрого доступа, организации каскадных меню. Цель стандартизации — облегчит пользователю работу с приложением. Надо, чтобы пользователю не приходилось думать, в каком меню и как ему надо открыть или сохранить файл, как ему получить справку, как работать с буфером обмена Clipboard и т.д. Для осуществления всех этих операций у пользователя, порабовавшего хотя бы с несколькими приложениями Windows, вырабатывается стойкий автоматизм действий и недопустимо этот автоматизм ломать.

Начнем рассмотрение требований с размещения заголовков меню. Типичная полоса главного меню приведена на рис. 3.5 в разделе 3.2.4 главы 3. Конечно, состав меню зависит от конкретного приложения. Но размещение общепринятых разделов должно быть стандартизированным. Все пользователи уже привыкли, что меню Файл размещается слева в полосе главного меню, раздел справки — справа, перед ним в приложениях MDI размещается меню Окно и т.д. Главное меню должно также снабжаться инструментальной панелью (см. рис. 3.5), быстрые кнопки которой дублируют наиболее часто используемые команды меню. На этих кнопках надо использовать, по возможности, привычные картинки. В C++Builder имеется множество изображений кнопок на все случаи жизни, но, к сожалению, они не всегда имеют привычный для пользователя рисунок.

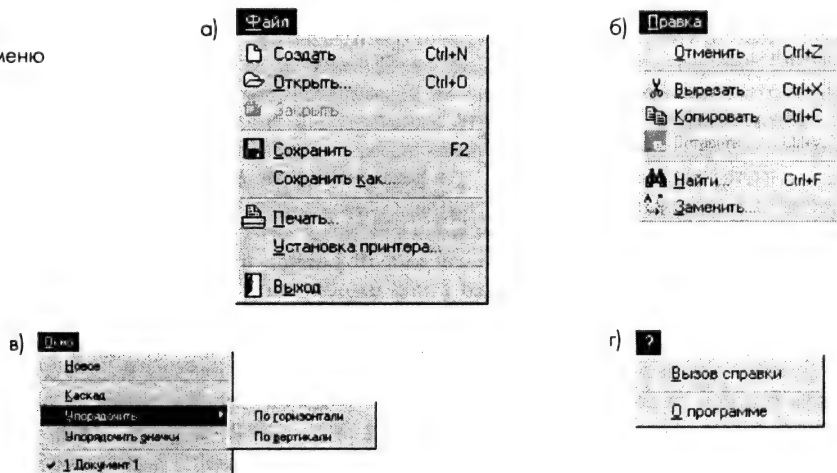
По возможности стандартным должно быть и расположение разделов в выпадающих меню. На рис. 4.3 приведены распространенные варианты меню Файл — работа с файлами, Правка — работа с текстами, Окно — управление окнами в приложении MDI и меню Справка или ? — просмотр справочных данных. Обратите внимание, что, например, раздел Выход всегда размещается последним в меню Файл, а раздел информации о версии программы О программе... — последним в справочном меню.

Группы функционально связанных разделов отделяются в выпадающих меню разделителями.

Названия разделов меню должны быть привычными пользователю. Если вы не знаете, как назвать какой-то раздел, не изобретайте свое имя, а попытайтесь найти аналогичный раздел в какой-нибудь русифицированной программе Microsoft для Windows. Названия должны быть краткими и понятными. Не используйте фраз, да и вообще больше двух слов, поскольку это перегружает экран и замедляет выбор пользователя. Названия разделов должны начинаться с заглавной буквы. Применительно к английским названиям разделов существует требование, чтобы каждое слово тоже начиналось с заглавной буквы. Но применительно к русским названиям это правило не применяется.

Рис. 4.3

Типовые меню



Названия разделов меню, связанных с вызовом диалоговых окон, должны заканчиваться многоточием, показывающим пользователю, что при выборе этого раздела ему предстоит установить в диалоге еще какие-то параметры.

Разделы, к которым относятся каскадные меню (например, раздел Упорядочить на рис. 4.3 в) должны заканчиваться стрелкой, указывающей на наличие дочернего меню данного раздела. C++Builder ставит эту стрелку автоматически, так что о ней вам думать не приходится. Вообще злоупотреблять каскадными меню не следует, так как пользователю не так просто до них добираться. Если в дочернем меню должно быть много разделов, например, связанных с какими-то опциями и настройками, то подумайте, не лучше ли вместо этого дочернего меню предусмотреть диалоговое окно, в котором эти опции будут более обозримыми и доступными.

В каждом названии раздела меню должен быть выделен подчеркиванием символ, соответствующий клавише быстрого доступа к разделу (клавиша Alt плюс подчеркнутый символ). Хотя вряд ли такими клавишами часто пользуются, но традиция указания таких клавиш незыблема.

Многим разделам могут быть поставлены в соответствие «горячие» клавиши, позволяющие обратиться к команде данного раздела, даже не заходя в меню. Комбинации таких «горячих» клавиш должны быть традиционными. Например, команды вырезания, копирования и вставки фрагментов текста практически всегда имеют «горячие» клавиши Ctrl-X, Ctrl-C и Ctrl-V соответственно. Заданные сочетания клавиш отображаются в заголовках соответствующих разделов меню (см. рис. 4.3 а и 4.3 б).

Начиная с C++Builder 4 предусмотрена возможность включать в меню пиктограммы, соответствующие некоторым разделам (см. рис. 4.3 а и 4.3 б). Это не входит в обязательные требования к меню и использование пиктограмм — дело вкуса.

Некоторые разделы меню, соответствующие выбору каких-то настроек, могут содержать индикаторы, показывающие, выбран или нет данный раздел, и могут содержать радиокнопки. Способы создания подобных разделов меню см. в разделе 3.6.1.

Не все разделы меню имеют смысл в любой момент работы пользователя с приложением. Например, если в приложении не открыт ни один документ, то бессмысленно выполнять команды редактирования в меню Правка. Если в тексте документа ничего не изменялось, то бессмысленным является раздел этого меню Отменить, отменяющий последнюю команду редактирования. Такие меню и отдельные разделы должны делаться временно недоступными или невидимыми. Это осуществляется заданием значения **false** свойствам раздела **Enabled** или **Visible** соот-

ветственно. Различие между недоступными и невидимыми разделами в том, что недоступный раздел виден в меню, но отображается серым цветом, а невидимый раздел просто исчезает из меню, причем нижележащие разделы смыкаются, занимая его место. Выбор того или иного варианта — дело вкуса и удобства работы пользователя. Вероятно, целиком меню лучше делать невидимыми, а отдельные разделы — недоступными. Например, пока ни один документ не открыт, меню Провка можно сделать невидимым, чтобы он не отвлекал внимания пользователя. А раздел Отменить этого меню в соответствующих ситуациях лучше делать недоступным, чтобы пользователь видел, что такой раздел в меню есть и им можно будет воспользоваться в случае ошибки редактирования.

Вот и все основные требования к главному меню приложения. Проектирование меню — не очень быстрый процесс и обидно повторять его снова и снова для каждого нового приложения, тем более, что требование стандартизации приводит к тому, что одни и те же разделы с одинаковыми свойствами кочуют из приложения в приложение. Поэтому можно рекомендовать один раз потратить время, создать меню, содержащее большинство разделов, которые могут вам понадобиться в различных приложениях, и сохранить это меню как шаблон (в разделе 3.6.1 рассказано, как это делается с помощью команды Save As Template в Конструкторе Меню). В дальнейшем вы сможете использовать этот шаблон в любом своем приложении, загружая его командой Insert From Template Конструктора Меню в компонент **MainMenu**. А удалить после этого разделы, не используемые в данном приложении, не представляет никакого труда — вы просто выделяете соответствующий раздел в Конструкторе Меню и нажимаете клавишу Del.

Помимо главного меню в приложении обычно должны быть контекстные всплывающие меню отдельных компонентов, например, окон редактирования. Эти меню обычно дублируют команды главного меню, используемые при работе с данным компонентом. В разделе 3.6.2 рассказано, как проектируются всплывающие меню с помощью компонента **PopupMenu**. Всплывающие меню желательно не перегружать разделами, вводя в них только действительно часто используемые команды.

4.1.6.2 Методика проектирования меню и инструментальной панели

Теперь обсудим, как можно удовлетворить рассмотренные в разделе 4.1.6.1 требования к меню в приложениях C++Builder. Отдельные вопросы, связанные с этим, уже рассмотрены в главе 3: разработка меню — раздел 3.6.1, разработка инструментальной панели — разделы 3.7.5 и 3.7.6, компоненты организации управления приложением — раздел 3.9. Ниже все это объединено вместе при рассмотрении основных этапов разработки полноценного меню. В качестве примера рассмотрим приложение, изображенное ранее на рис. 3.5. Его форма приведена на рис. 4.4.

Рис. 4.4

Форма приложения, изображенного ранее на рис. 3.5



Итак, основные этапы разработки сводятся к следующему:

1. Перенесите на форму компонент диспетчеризации действий **ActionList** (на рис. 4.4 он второй слева в верхнем ряду невизуальных компонентов). Применение этого компонента не обязательно. Но, привыкнув к работе с ним, вы увидите, что его применение экономит вам много времени и делает код более прозрачным и простым в сопровождении.
2. Перенесите на форму компонент **ImageList** — список изображений для разделов меню и быстрых кнопок инструментальной панели (на рис. 4.4 он первый слева в верхнем ряду).
3. С помощью редактора списка изображений, вызываемого двойным щелчком на **ImageList**, заполните список изображениями, символизирующими основные команды будущего меню (см. раздел 3.9.2 главы 3).
4. Перенесите на форму компоненты стандартных диалогов типа «Открыть файл» (компонент **OpenFileDialog**), «Сохранить файл как» (компонент **SaveFileDialog**) и др. На рис. 4.4 диалоги — все компоненты нижнего ряда. Перенесите также другие компоненты, к которым будут относиться действия. Например, на рис. 4.4 на форме расположено многострочное окно редактирования **RichEdit**.
5. Компонент диспетчеризации действий **ActionList** свяжите со списком изображений, установив его свойство **Images** равным **ImageList1** — имени компонента **ImageList**.
6. Двойным щелчком на компоненте **ActionList** вызовите редактор действий и занесите в нем все действия, соответствующие будущим разделам меню (см. рис. 3.59 в разделе 3.9.1 главы 3).
7. После занесения в список очередного действия выделите его в правом окне рис. 3.59 и на странице свойств окна Редактора Кода задайте его свойства. Прежде всего задайте **Name** (имя) — желательно осмысленное, связанное с введенным действием. Только не задавайте имена, которые могут оказаться тождественными именам каких-то функций, переменных и т.д. Чтобы не возникало подобных казусов, можно рекомендовать добавлять ко всем именам в начале символ «A», как это сделано на рис. 3.59. Вы должны также установить свойство **Caption** — надпись, которая далее будет появляться в кнопках, разделах меню и т.д. При этом надо следовать всем изложенным в разделе 4.1.6.1 требованиям: выделение подчеркнутого символа, многоточия в конце разделов, вызывающих диалоговые окна и т.д. Вы можете также задать свойства: **ShortCut** — быстрые клавиши (если они должны назначаться для данного действия), **Hint** — надписи на ярлычках подсказок и в строке состояний (см. подробнее в разделе 4.1.9), **HelpContext** — идентификатор темы контекстной справки (если для данного действия предусмотрена справка), **ImageIndex** — индекс изображения в **ImageList**, соответствующего данному действию. Можете также, при необходимости, задать свойства **Enabled** — доступность, **Checked** — занесение индикатора выделения и др.
8. Для каждого действия на странице событий Инспектора Объектов определено событие **OnExecute**. В обработчике этого события вам надо описать операции, соответствующие данному действию. Например, для действия **AExit** (выход из программы) обработчик может иметь вид

```
void __fastcall TForm1::AExitExecute(TObject *Sender)
{
    Close();
}
```

Обратите внимание, что заголовок процедуры (**AExitExecute**) получается осмысленным — он указывает на характер действия. Такое название процедуры, конечно, много понятнее, чем **Button5Click** или **N1Click** — имена, получающиеся без использования компонента **ActionList**.

9. После завершения списка действий перенесите на форму компонент **MainMenu** — главное меню (на рис. 4.4 третий компонент слева в верхнем ряду). Установите его свойство **Images** равным **ImageList1** — имени компонента **ImageList**. Вызовите двойным щелчком на **MainMenu** редактор меню (см. раздел 3.6.1). Вы можете не задавать для вводимых вами разделов меню никаких свойств. Достаточно, выделив очередную пунктирную рамку в редакторе меню, задать в Инспекторе Объектов для очередного раздела свойство **Action**. Для этого достаточно щелкнуть в строке правее этого свойства и из выпадающего списка, в котором имеются все введенные вами действия, выбрать нужное. Вы увидите, что при этом автоматически заполнятся все свойства данного раздела меню: появится надпись, изображение (если оно предусмотрено), тексты подсказок в свойстве **Hint** и т.д. На странице событий Инспектора Объектов вы увидите, что в событие **OnClick** (щелчок) уже занесена ссылка на ранее написанный вами для этого действия обработчик. Таким образом, применение компонента **ActionList** существенно упростило разработку меню.
10. Перенесите на форму компонент инструментальной панели **ToolBar**. Щелкая на нем правой кнопкой мыши выбирайте команды **New Button** (новая кнопка) или **New Separator** (новый разделитель). Для каждой новой кнопки так же, как ранее для разделов меню, устанавливайте свойство **Action**. Все остальные свойства перенесутся в кнопку автоматически.
11. Если необходимо обеспечить в приложении контекстные всплывающие меню, то перенесите на форму один или несколько (по числу различных контекстных меню) компонентов **PopupMenu** — на рис. 4.4 это второй справа компонент в верхнем ряду. Свяжите их с компонентом **ImageList**, задав свойство **Images**. Далее, сделав двойной щелчок на **PopupMenu**, вы попадете в тот же редактор меню, что и в случае **MainMenu**. Вы можете проектировать всплывающее меню так же, как главное. Но, поскольку всплывающее меню обычно дублирует какие-то разделы уже сформированного главного меню, то можно обойтись копированием соответствующих разделов. Техника подобного копирования рассмотрена в разделе 3.6.2.
12. Когда контекстное меню сформировано, для привязки его к необходимому компоненту выделите этот компонент на форме и в инспекторе объектов задайте его свойство **PopupMenu** равным имени компонента соответствующего **PopupMenu**.
13. Можете добавить на форму панель состояния **StatusBar** (на рис. 4.4, внизу) и компонент **ApplicationEvents** (на рис. 4.4 он правый в верхнем ряду). Эти компоненты могут обеспечить отображение в панели состояния развернутых подсказок (см раздел 4.1.8).

Вот в общих чертах последовательность операций при проектировании меню и инструментальной панели с использованием компонентов **ActionList**, **ImageList**, **MainMenu**, **PopupMenu** и **ToolBar**. Как видите, это не очень быстрый процесс и обидно повторять его снова и снова для каждого нового приложения, тем более, что требование стандартизации приводит к тому, что одни и те же разделы с одинаковыми свойствами кочуют из приложения в приложение. Поэтому можно рекомендовать один раз потратить время, создать меню, содержащее большинство разделов, которые могут вам понадобиться в различных приложениях, и затем сохранить его в качестве шаблона. Вопросы создания шаблонов рассмотрены в главе 7 в разделе 7.2.

4.1.7 Компоновка форм

Каждое окно, которое вы вводите в свое приложение, должно быть тщательно продумано и компоновано. Удачная компоновка может стимулировать эффективную работу пользователя, а неудачная — рассеивать внимание, отвлекать, заставлять тратить лишнее время на поиск нужной кнопки или индикатора.

Управляющие элементы и функционально связанные с ними компоненты экрана должны быть зрительно объединены в группы, заголовки которых коротко и четко поясняют их назначение. Такое объединение позволяет осуществлять различные панели, рассмотренные в разделе 3.7. Можно рекомендовать, как правило, размещать компоненты не непосредственно на форме, а на таких панелях. Но и внутри панелей надо продумывать размещение компонентов как с точки зрения эстетики, так и с точки зрения визуального отражения взаимоотношений элементов. Например, если имеется кнопка, которая разворачивает окно списка, то эти два компонента должны быть визуально связаны между собой: размещены на одной панели и в непосредственной близости друг от друга. Если же ваш экран представляет собой случайные скопления кнопок, то именно так он и будет восприниматься. И в следующий раз пользователь не захочет пользоваться вашей программой.

Каждое окно должно иметь некоторую центральную тему, которой подчиняется его композиция. Пользователь должен понимать, для чего предназначено данное окно и что в нем наиболее важно. При этом недопустимо перегружать окно большим числом органов управления, ввода и отображения информации. В окне должно отображаться главное, а все детали и дополнительную информацию можно отнести на вспомогательные окна. Для этого полезно вводить в окно кнопки с надписью *Больше...*, многоточие в которой показывает, что при нажатии этой кнопки откроется вспомогательное окно с дополнительной информацией. Помогают также разгрузить окно многостраничные компоненты с закладками, рассмотренные в разделе 3.7.4. Они дают возможность пользователю легко переключаться между разными по тематике страницами, на каждой из которых имеется необходимый минимум информации. Примеры удачной организации окон вы можете посмотреть в C++Builder, выполнив команду *Tools | Environment Options* и полистав страницы окна опций.

Еще один принцип, которого надо придерживаться при проектировании окон — стилистическое единство всех окон в приложении. Недопустимо, чтобы сходные по функциям органы управления в разных окнах назывались по-разному или размещались в разных местах окон. Все это мешает работе с приложением, отвлекает пользователя, заставляет его думать не о сущности работы, а о том, как приспособиться к тому или иному окну. Появившийся в C++Builder 5 компонент **Frame** — фрейм (см. раздел 3.7.8) позволяет один раз разработать некий повторяющийся фрагмент окна, поместить его в Депозитарий (см. главу 7 раздел 7.4), а затем использовать его в разных формах и приложениях.

Стилистическому единству окон приложения способствует имеющаяся в C++Builder возможность создания иерархии форм. Эта возможность рассмотрена в разделе 7.4. Приступая к большому проекту полезно сначала создать иерархическую последовательность форм, а затем уже, используя эту иерархию, проектировать конкретные окна. Кроме всего прочего, это позволяет сэкономить немало времени при разработке, потому что внесение каких-то усовершенствований в одну форму автоматически ведет к тиражированию этих изменений в других формах.

Единство стилистических решений важно не только внутри приложения, но и в рамках серии разрабатываемых вами приложений. Это нетрудно обеспечить с помощью имеющихся в C++Builder многочисленных способов повторного использования кодов (см. главу 7). Вам достаточно один раз разработать какие-то часто применяемые формы — ввода пароля, запроса или предупреждения пользователя и т.п., включить их в депозитарий, а затем вы можете использовать их многократно во всех своих проектах.

4.1.8 Последовательность фокусировки элементов

При проектировании приложения важно правильно определить последовательность табуляции оконных компонентов. Под этим понимается последовательность, в которой переключается фокус с компонента на компонент, когда пользователь нажимает клавишу табуляции **Tab**. Это важно, поскольку в ряде случаев пользователю удобнее работать не с мышью, а с клавиатурой. Пусть, например, вводя данные о каком-то сотруднике пользователь должен в отдельных окнах редактирования указать фамилию, имя и отчество. Конечно, набрав фамилию ему удобнее нажать клавишу **Tab** и набирать имя, а потом опять, нажав **Tab**, набирать отчество, чем каждый раз отрываться от клавиатуры, хватать мышь и переключаться в новое окно редактирования.

Свойство формы **ActiveControl**, установленное в процессе проектирования, определяет, какой из размещенных на форме компонентов будет в фокусе в первый момент при выполнении приложения. В процессе выполнения это свойство изменяется и показывает тот компонент, который в данный момент находится в фокусе.

Последовательность табуляции задается свойствами **TabOrder** компонентов. Первоначальная последовательность табуляции определяется просто той последовательностью, в которой размещались управляющие элементы на форме. Первому элементу присваивается значение **TabOrder**, равное 0, второму 1 и т.д. Значение **TabOrder**, равное нулю, означает, что при первом появлении формы на экране в фокусе будет именно этот компонент (если не задано свойство формы **ActiveControl**). Если на форме имеются панели, фреймы и иные контейнеры, включающие в себя другие компоненты, то последовательность табуляции составляется независимо для каждого компонента-контейнера. Например, для формы будет последовательность, включающая некоторую панель как один компонент. А уже внутри этой панели будет своя последовательность табуляции, определяющая последовательность получения фокуса ее дочерними оконными компонентами.

Каждый управляющий элемент имеет уникальный номер **TabOrder** внутри своего родительского компонента. Поэтому изменение значения **TabOrder** какого-то элемента на значение, уже существующее у другого элемента, приведет к тому, что значения **TabOrder** всех последующих элементов автоматически изменятся, чтобы не допустить дублирования. Если задать элементу значение **TabOrder**, большее, чем число элементов в родительском компоненте, он просто станет последним в последовательности табуляции.

Из-за того, что при изменении **TabOrder** одного компонента могут меняться **TabOrder** других компонентов, устанавливать эти свойства поочередно в отдельных компонентах трудно. Поэтому в среде проектирования **C++Builder 5** имеется специальная команда **Edit | Tab Order**, позволяющая в режиме диалога задать последовательность табуляции всех элементов.

Значение свойства **TabOrder** играет роль только, если другое свойство компонента — **TabStop** установлено в **true** и если компонент имеет родителя. Например, для формы свойство **TabOrder** имеет смысл только в случае, если для формы задан родитель в виде другой формы. Установка **TabStop** в **false** приводит к тому, что компонент выпадает из последовательности табуляции и ему невозможно передать фокус клавишей **Tab** (однако предать фокус мышью, конечно, можно).

Имеется и программная возможность переключения фокуса — это метод **SetFocus**. Например, если вы хотите переключить в какой-то момент фокус на окно **Edit2**, вы можете сделать это оператором:

```
Edit2->SetFocus();
```

Выше говорилось, что в приложении с несколькими окнами редактирования, в которые пользователь должен поочередно вводить какие-то данные, ему удобно переключаться между окнами клавишей табуляции. Но еще удобнее пользователю, закончившему ввод в одном окне, нажать клавишу **Enter** и автоматически пе-

рейти к другому окну. Это можно сделать, обрабатывая событие нажатия клавиши **OnKeyDown** (см. о событиях при нажатии клавиш в разделе 4.3.2). Например, если после ввода данных в окно **Edit1** пользователю надо переключаться в окно **Edit2**, то обработчик события **OnKeyDown** окна **Edit1** можно сделать следующим:

```
void __fastcall TForm1::Edit1KeyDown(TObject *Sender,
                                     WORD &Key, TShiftState Shift)
{
    if (Key == VK_RETURN) Edit2->SetFocus();
}
```

Единственный оператор этого обработчика проверяет, не является ли клавиша, нажатая пользователем, клавишей Enter. Если это клавиша Enter, то фокус передается окну **Edit2** методом **SetFocus**.

Можно подобные операторы ввести во все окна, обеспечивая требуемую последовательность действий пользователя (впрочем, свобода действий пользователя от этого не ограничивается, поскольку он всегда может вернуться вручную к нужному окну). Однако можно сделать все это более компактно, используя один обработчик для различных окон редактирования и кнопок. Для этого может использоваться метод **FindNextControl**, возвращающий дочерний компонент, следующий в последовательности табуляции. Если компонент, имеющий в данный момент фокус, является последним в последовательности табуляции, то возвращается первый компонент этой последовательности. Метод определен следующим образом:

```
TWinControl* __fastcall FindNextControl(
    TWinControl* CurControl,
    bool GoForward, bool CheckTabStop,
    bool CheckParent);
```

Он находит и возвращает следующий за указанным в параметре **CurControl** дочерний оконный компонент в соответствии с последовательностью табуляции.

Параметр **GoForward** определяет направление поиска. Если он равен **true**, то поиск проводится вперед и возвращается компонент, следующий за **CurControl**. Если же параметр **GoForward** равен **false**, то возвращается предшествующий компонент.

Параметры **CheckTabStop** и **CheckParent** определяют условия поиска. Если **CheckTabStop** равен **true**, то просматриваются только компоненты, в которых свойство **TabStop** установлено в **true**. При **CheckTabStop** равно **false** значение **TabStop** не принимается во внимание. Если параметр **CheckParent** равен **true**, то просматриваются только компоненты, в свойстве **Parent** которых указан данный оконный элемент, т.е. просматриваются только прямые потомки. Если **CheckParent** равен **false**, то просматриваются все, даже косвенные потомки данного элемента.

Таким образом, для решения поставленной нами задачи — автоматической передачи фокуса при нажатии клавиши Enter, вы можете написать единый обработчик событий **OnKeyDown** всех интересующих вас оконных компонентов, содержащий оператор:

```
if (Key == VK_RETURN)FindNextControl(
    (TWinControl *)Sender, true, true, false)->SetFocus();
```

Этот оператор обеспечивает передачу фокуса очередному компоненту. Для кнопок аналогичный оператор можно вставить в обработчик события **OnClick**:

```
FindNextControl((TWinControl *)Sender, true, true, false)->SetFocus();
```

В приведенных кодах используется параметр **Sender**, передаваемый во все обработчики событий и соответствующий тому компоненту, в котором произошло событие. Этот параметр имеет тип **TObject**. А поскольку функция **FindNextControl** требует параметр объекта типа **TWinControl**, необходимо явным образом привести тип **TObject** к производному от него типу **TWinControl** с помощью операции

(TWinControl *)Sender (см. подробнее о работе с параметром **Sender** в разделе 1.5.6.2 главы 1).

4.1.9 Подсказки и контекстно-зависимые справки

Приложение должно предельно облегчать работу пользователя, снабжая его системой подсказок, помогающих сориентироваться в приложении. Эта система включает в себя:

- Ярлычки, которые всплывают, когда пользователь задержит курсор мыши над каким-то элементом окна приложения. В частности, такими ярлычками обязательно должны снабжаться быстрые кнопки инструментальных панелей, поскольку нанесенные на них пиктограммы часто не настолько выразительны, чтобы пользователь без дополнительной подсказки мог понять их назначение.
- Кнопка справки в полосе заголовка окна, позволяющая пользователю посмотреть во всплывающих окнах назначение различных элементов окна.
- Более развернутые подсказки в панели состояния или в другом отведенном под это месте экрана, которые появляются при перемещении курсора мыши в ту или иную область окна приложения.
- Встроенную систему контекстно-зависимой оперативной справки, вызываемую по клавише F1.
- Раздел меню Справка, позволяющий пользователю открыть стандартный файл справки Windows .hlp, содержащий в виде гипертекста развернутую информацию по интересующим пользователя вопросам.

Тексты ярлычков и подсказок панели состояния устанавливаются для любых визуальных компонентов в свойстве **Hint** в виде строки текста, состоящей из двух частей, разделенных символом вертикальной черты '|'. Первая часть, обычно очень краткая, предназначена для отображения в ярлычке; вторая более развернутая подсказка предназначена для отображения в панели состояния или ином заданном месте экрана. Например, в кнопке, соответствующей команде открытия файла, в свойстве **Hint** может быть задан текст:

Открыть|Открытие текстового файла

Как частный случай, в свойстве **Hint** может быть задана только первая часть подсказки без символа '|'.

Для того, чтобы первая часть подсказки появлялась во всплывающем ярлычке, когда пользователь задержит курсор мыши над данным компонентом, надо сделать следующее:

1. Указать тексты свойства **Hint** для всех компонентов, для которых вы хотите обеспечить ярлычок подсказки.
2. Установить свойства **ShowHint** (показать подсказку) этих компонентов в **true** или установить в **true** свойство **ParentShowHint** (отобразить свойство **ShowHint** родителя) и установить в **true** свойство **ShowHint** контейнера, содержащего данные компоненты.

Конечно, вы можете устанавливать свойства в **true** или **false** программно, включая и отключая подсказки в различных режимах работы приложения.

При **ShowHint**, установленном в **true**, окно подсказки будет всплывать, даже если компонент в данный момент недоступен (свойство **Enabled** = **false**).

Если вы не задали значение свойства компонента **Hint**, но установили в **true** свойство **ShowHint** или установили в **true** свойство **ParentShowHint**, а в родительском компоненте **ShowHint** = **true**, то в окне подсказки будет отображаться текст **Hint** из родительского компонента.

Правда, все описанное выше справедливо при значении свойства **ShowHint** приложения **Application** равном **true** (это значение задано по умолчанию). Если

установить **Application.ShowHint** в **false**, то окна подсказки не будут появляться независимо от значений **ShowHint** в любых компонентах.

Свойства **Hint** компонентов можно также использовать для отображения текстов заключенных в них сообщений в какой-то метке или панели с помощью функций **GetShortHint** и **GetLongHint**, первая из которых возвращает первую часть сообщения, а вторая — вторую (если второй части нет, то возвращается первая часть). Например, эти функции можно использовать в обработчиках событий **OnMouseMove**, соответствующих прохождению курсора мыши над данным компонентом. Так обработчик

```
voidfastcall TForm1::Button1MouseMove(TObject *Sender,
                                     TShiftState Shift, int X, int Y)
{
    TControl *Send = (TControl *)Sender;
    Panel1->Caption = GetShortHint(Send->Hint);
    Panel2->Caption = GetLongHint(Send->Hint);
}
```

отобразит в панели **Panel1** первую, а в панели **Panel2** — вторую часть свойства **Hint** всех компонентов, над которыми будет перемещаться курсор, если в этих компонентах в событии **OnMouseMove** указан этот обработчик **Label1MouseMove**. Причем это не зависит от значения их свойства **ShowHint**.

Еще один пример. Пусть вы хотите, чтобы при нажатии некоторой кнопки **Button1** вашего приложения в панели **Panel1** высвечивалась подсказка пользователю, например, «Укажите имя файла», а сама кнопка имела всплывающий ярлычок подсказки с текстом «Ввод». Тогда вы можете задать свойству **Hint** этой кнопки значение «Ввод|Укажите имя файла», задать значение **true** свойству **ShowHint**, а в обработчик события нажатия этой кнопки вставить оператор

```
Panel1->Caption = GetLongHint(Button1->Hint);
```

Если же вы не хотите отображать ярлычок подсказки для кнопки, то можете ограничиться значением **Hint**, равным «Укажите имя файла», а приведенный выше оператор оставить неизменным или заменить на эквивалентный ему в данном случае оператор

```
Panel1->Caption = GetShortHint(Button1->Hint);
```

или даже просто на оператор

```
Panel1->Caption = Button1->Hint;
```

Перед тем моментом, когда должен отобразиться ярлычок какого-нибудь компонента, возникает событие приложения **OnShowHint**. В обработчике этого события можно организовать какие-то дополнительные действия, например, изменить отображаемый текст. Особенно легко работать с событиями приложения в **C++Builder 5**, в котором имеется компонент **ApplicationEvents**, перехватывающий все эти события (см. подробнее в разделе 3.9.3). В обработчик его события **OnShowHint** можно поместить те операторы, которые надо выполнить перед отображением ярлычка. Заголовок этого обработчика имеет вид:

```
void __fastcall TForm1::ApplicationEvents1ShowHint(
    AnsiString &HintStr, bool &CanShow,
    THintInfo &HintInfo)
```

Здесь передаваемый по ссылке параметр **HintStr** — отображаемый в ярлычке текст. В обработчике этот текст можно изменить. Так же по ссылке передается параметр **CanShow**. Если в обработчике установить его равным **false**, то ярлычок отображаться не будет. Третий параметр, передаваемый по ссылке — **HintInfo**. Это структура, поля которой содержат информацию о ярлычке: его координаты, цвет, задержки появления и т.п. В частности, имеется поле **HintControl** — компонент, сообщение которого должно отображаться в ярлычке, и поле **HintStr** — отобра-

жаемое сообщение. По умолчанию **HintInfo.HintStr** — первая часть свойства **Hint** компонента. Но в обработчике это значение можно изменить. В разделах 3.7.8 и 3.9.3 приведены примеры использования обработчика события **OnShowHint** для отображения в ярлычке длинных текстов, содержащихся в окнах редактирования.

Имеется еще один способ отображения второй части сообщения, записанного в **Hint**, в строке состояния или какой-то области экрана в моменты, когда курсор мыши проходит над компонентом — это использование обработки события приложения **OnHint**. Это событие не того компонента, над которым проходит курсор мыши, а именно приложения — объекта **Application**. В C++Builder 5 это событие также перехватывается компонентом **ApplicationEvents**. Если обработчик этого события определен, то в момент прохождения курсора над компонентом, в котором задано свойство **Hint**, вторая часть сообщения компонента заносится в свойство **Hint** объекта **Application**. Если свойство **Hint** компонента содержит только одну часть, то в свойство **Hint** объекта **Application** заносится эта первая часть.

Если ваше приложение содержит инструментальную панель с быстрыми кнопками, то, как правило, эти кнопки должны снабжаться не только всплывающими ярлычками, но и развернутыми подсказками в панели состояния (см. пример панели на рис. 4.4 и 3.5). Если у вас есть подобный пример, вы можете опробовать на нем методику отображения подсказок в панели состояния. А можете взять какой-нибудь более простой пример. Для реализации подсказок в панели состояния надо перенести на форму панель состояния — компонент **StatusBar** со страницы Win32 (см. раздел 3.7.7) и компонент **ApplicationEvents**. Если вам нужна односекционная панель, установите свойство **SimplePanel** панели **StatusBar** в **true**. Если вы хотите использовать многосекционную панель, то в приведенном ниже операторе надо заменить свойство **SimpleText** ссылкой на панель, в которой вы хотите отображать подсказку.

Напишите какие-нибудь тексты в свойствах **Hint** компонентов, размещенных на вашей форме. А в обработчик события **OnHint** компонента **ApplicationEvents** вставьте оператор

```
StatusBar1->SimpleText = Application->Hint;
```

Запустите приложение на выполнение. Вы увидите, что тексты подсказок отображаются в панели состояния, когда курсор мыши перемещается над тем или иным окном редактирования. Причем это не мешает появляться ярлычкам, отображающим тексты окон.

Более подробные пояснения пользователю может дать контекстно-зависимая справка, встроенная в приложение. Она позволяет пользователю нажать в любой момент клавишу F1 и получить развернутую информацию о том компоненте в окне, который в данный момент находится в фокусе. Для того, чтобы это осуществить, надо разработать для своего приложения файл справки **.help**. Как это сделать, подробно описано в главе 8. Затем надо в компонентах, для которых вы хотите обеспечить контекстно-зависимую справку, установить в свойстве **HelpContext** номер контекстной справки, соответствующей данному компоненту. Если **HelpContext** компонента равен нулю, то данный компонент наследует это свойство от своего родительского компонента. Например, для всех компонентов, размещенных на некоторой панели можно задать **HelpContext** = 0, а для самой панели задать отличное от нуля значение **HelpContext**, соответствующее теме, описывающей назначение всех компонентов панели.

Для того, чтобы все это работало, надо выполнить команду Project | Options и в окне Project Options (опции проекта) на странице Application (приложение) установить значение опции Help file, равное имени подготовленного файла **.hlp**. Это приведет к тому, что в головном файле проекта появится оператор вида:

```
Application->HelpFile = "<имя файла>.hlp";
```

В этом операторе используется метод **HelpFile**, определяющий файл справки, к которому обращается проект. Этот метод и подобный оператор можно использовать в приложении в любом обработчике события, если в какие-то моменты требуется сменить используемый файл справки.

Если предполагается, что файл справки будет расположен в том же каталоге, где находится само приложение, то имя файла и в окне Опции проекта, и в приведенном выше операторе надо задавать без указания пути. Иначе приложение, работающее на вашем компьютере, перестанет работать на компьютере пользователя, у которого каталоги не совпадают с вашими.

Для того, чтобы приложение в свойствах **HelpContext** могло ссылаться на какой-то номер контекстной справки, в файле проекта справки **.hlp** в разделе **[MAP]** надо поместить таблицу соответствия использованных значений **HelpContext** и тем файла **.hlp**. Все необходимые для этого операции подробно рассмотрены в главе 8.

В заключение поговорим о традиционном разделе меню Справка, позволяющем пользователю открыть файл справки **.hlp** и получить развернутую информацию по всем вопросам, связанным с данным приложением. В обработчик события при выборе данного раздела меню или при нажатии соответствующих кнопок помощи вставляются операторы вида

```
Application->HelpContext(<номер темы>);
```

Задаваемые в этих операторах номера тем аналогичны используемым при задании свойств **HelpContext**. Это номер той темы, которая первой отобразится при открытии окна справки. А в дальнейшем пользователь, как обычно, может перейти, работая с программой справки, к любой интересующей его теме.

Имеется еще несколько методов объекта **Application**, обеспечивающих работу со справочными файлами. Все они подробно рассмотрены в разделе 3.9.3.

Еще один механизм подсказок, связанный с файлом справок **.hlp** — кнопка справки, присутствующая в заголовках многих современных окон Windows. Нажав ее, пользователь может подвести курсор мыши, изменивший свою форму на вопросительный знак, к какому-то компоненту, щелкнуть и во всплывшем окне появится развернутая подсказка, поясняющая назначение данного компонента. Для того, чтобы ввести такую возможность в свое приложение, надо установить в **true** подсвойство **byHelp** свойства **BorderIcons** вашей формы. Однако, не для всех форм это вызовет появление в заголовке окна кнопки справки. Кнопка появится только в случае, если свойство **BorderStyle** формы установлено в **bsDialog**. Никакого программирования работа с кнопкой справки не требует. Все будет выполняться автоматически. Достаточно предусмотреть в файле **.hlp** соответствующие темы и сделать на них ссылки в свойствах **HelpContext** компонентов. Эти темы будут появляться при соответствующих действиях пользователя во всплывающих окнах.

4.2 Проектирование окон с изменяемыми размерами

4.2.1 Выравнивание компонентов — свойство Align

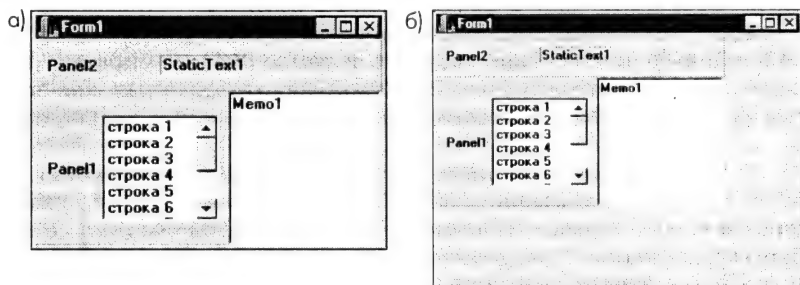
Если проектируется окно, размеры которого пользователь может изменять во время выполнения приложения, то необходимо принять меры, чтобы компоненты в окне при этом тоже изменяли свои размеры или местоположение, равномерно распределяясь по площади окна и не оставляя пустых мест.

Пусть, например, вы проектируете форму, окно которой содержит панель **Panel1**, на которой будут размещаться какие-то управляющие компоненты и спи-

сок **ListBox1**, панель **Panel2**, в середине которой будет размещаться некоторая надпись в метке **StaticText1**, и компонент **Memo1**, в котором будут редактироваться тексты (рис. 4.5 а). Если, разместив все это на форме, вы не примете мер для того, чтобы при изменении размеров окна компоненты изменялись, то при том размере формы, который вы проектировали, все будет выглядеть нормально (рис. 4.5 а). Но если пользователь растянет размеры формы, надеясь увеличить площадь редактирования и длину списка, то приложение приобретет нелепый вид, показанный на рис. 4.5 б. Увеличение формы просто приводит к увеличению на ней пустого места.

Рис. 4.5

Результаты изменения размеров окна при неправильном проектировании

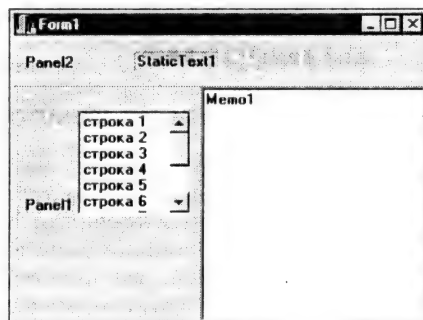


Чтобы избежать таких неприятностей, у многих компонентов и, в частности, у панелей, есть свойство **Align** — выравнивание. По умолчанию оно равно **alNone**, что означает, что никакое выравнивание не осуществляется. Но его можно задать равным **alTop**, или **alBottom**, или **alLeft**, или **alRight**, что будет означать, что компонент должен занимать всю верхнюю, или нижнюю, или левую, или правую часть клиентской области компонента-контейнера. Под клиентской областью понимается вся свободная площадь формы или другого контейнера, в которой могут размещаться включенные в этот контейнер компоненты. Можно также задать свойству **Align** компонента значение **alClient**, что приводит к заполнению компонентом всей свободной клиентской области. Во всех этих случаях размеры компонента будут автоматически изменяться при изменении размеров контейнера.

В приведенном выше примере логично для панели **Panel2** задать значение **Align**, равным **alTop**, чтобы ширина панели автоматически менялась с изменением ширины окна. Панель займет всю верхнюю часть формы, а ее высота будет такой, какую вы установите. Для панели **Panel1** следует задать значение **Align**, равным **alLeft**, так как увеличение ее ширины не имеет смысла, а увеличение высоты позволяет увидеть большую часть списка **ListBox1**. Компонент **Panel1** займет всю левую часть клиентской области, оставшейся свободной после размещения **Panel2**. Для компонента **Memo1** следует задать значение **Align**, равным **alClient**, что позволит компоненту увеличиваться и в высоту, и в ширину, увеличивая площадь редактирования. При этом компонент **Memo1** займет всю площадь клиентской области, оставшуюся после размещения **Panel1** и **Panel2**. В результате во время вы-

Рис. 4.6

Изменение размеров панелей при использовании выравнивания



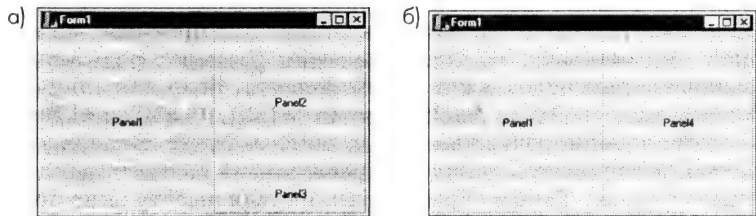
полнения при изменении пользователем размеров окна приложение будет иметь вид, представленный на рисунке 4.6.

Задавать компонентам соответствующие значения свойства **Align** в приведенном примере надо именно в указанной выше последовательности: **alTop** для **Panel2**, **alLeft** для **Panel1**, **alClient** для **Memo1**. Если, например, начать с задания **alClient** для **Memo1**, то компонент **Memo1** займет всю площадь формы и остальные компоненты вообще не будут видны.

Значения свойства **Align** имеют определенный приоритет: значения **alTop** и **alBottom** имеют более высокий приоритет, чем значения **alLeft** и **alRight**. Поэтому не любые варианты размещения и выравнивания панелей можно реализовать непосредственно. Например, пусть вы хотите создать форму, вид которой приведен на рис. 4.7 а. Причем вам надо, чтобы при изменении размеров окна панель **Panel1** продолжала занимать всю левую часть формы (т.е. надо задать **Align = alLeft**), **Panel3** занимала бы всю нижнюю часть площади, свободной от **Panel1** (т.е. для **Panel3** надо задать **Align = alBottom**), а **Panel2** занимала бы всю оставшуюся часть клиентской области (т.е. для **Panel2** надо задать **Align = alClient**). Напрямую все это сделать невозможно. Если вы задали для **Panel1** значение **alLeft**, а затем для **Panel3** зададите значение **alBottom**, то в силу приоритета **alBottom** панель **Panel3** займет всю нижнюю часть формы, вытеснив оттуда **Panel1**.

Рис. 4.7

Проектирование с учетом приоритетов значений **Align**



В подобных случаях приходится вводить дополнительные панели-контейнеры. В приведенном примере на форме сначала надо разместить **Panel1** и дополнительную панель **Panel4**, как показано на рис. 4.7 б. Для **Panel1** задается **alLeft**, а для **Panel4** задается **alClient**. Затем панели **Panel2** и **Panel3** размещаются на **Panel4** и для них задаются значения: **alBottom** для **Panel3** (поскольку клиентской областью для **Panel3** является область панели **Panel4**, то **Panel3** займет нижнюю часть правой половины экрана) и значение **alClient** для **Panel2**. В результате получим нужное выравнивание всех панелей.

4.2.2 Изменение местоположения и размеров компонентов

Заданием значений **Align** задачу планировки окна с изменяющимися размерами еще нельзя считать решенной. Вернемся к ранее рассмотренному примеру. Рис. 4.6 показал, что в нем при изменении размеров окна метка **StaticText1**, остающаяся на месте, перестает быть в середине панели, а размер списка **ListBox1** не изменяется.

Чтобы устранить эти недостатки, можно воспользоваться свойствами компонента, определяющими его привязку, местоположение и размеры.

Оконные компоненты имеют свойство **Anchors** — привязку к родительскому компоненту при изменении размеров последнего. Это свойство представляет собой множество, которое может содержать следующие элементы:

akTop	Верхний край компонента привязан к верхнему краю родительского компонента
akLeft	Левый край компонента привязан к левому краю родительского компонента
akRight	Правый край компонента привязан к правому краю родительского компонента
akBottom	Нижний край компонента привязан к нижнему краю родительского компонента

Если в множестве **Anchors** присутствуют привязки к противоположным сторонам родительского компонента, то при изменении размеров родительского компонента происходит растяжение или сжатие дочернего компонента, поскольку расстояния от сторон родительского компонента выдерживаются. Сжатие может происходить вплоть до полного исчезновения изображения данного компонента. Поэтому необходимо ограничивать диапазон допустимого изменения размеров. Способы такого ограничения будут рассмотрены в разделе 4.2.4.

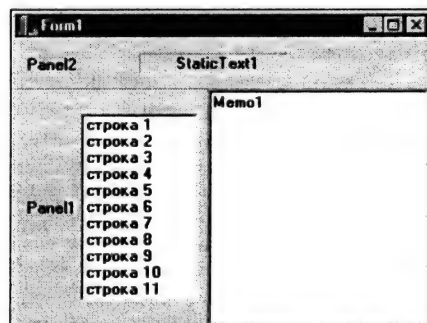
По умолчанию привязка осуществляется к левому и верхнему краям родительского компонента. Т.е. по умолчанию свойство **Anchors** равно **[akLeft,akTop]**. Если задать в свойстве **Anchors** привязку к противоположным сторонам родительского компонента, то при изменении размеров родительского компонента будут меняться размеры и данного компонента. Если для списка **ListBox1** в нашем примере задать свойство **Anchors** равным **[akLeft,akTop,akBottom]**, то при изменении высоты панели, содержащей этот список, будут поддерживаться постоянными расстояния верхнего и нижнего краев списка соответственно от верхнего и нижнего краев панели. Таким образом, увеличивая высоту окна пользователь может увеличивать число строк, видимых в списке без прокрутки. Если задать свойство **Anchors** равное **[akLeft,akTop, akRight]** для метки **StaticText1**, то при изменении ширины содержащей ее панели будут поддерживаться постоянными расстояния левого и правого краев метки от соответствующих краев панели. Метка будет оставаться в центре панели, но ее размер будет изменяться. Вариант нашего примера с заданными описанным способом свойствами **Anchors** для **ListBox1** и **StaticText1**, приведен на рис. 4.8. Как видно из него, теперь приложение сохраняет нормальный вид и при увеличении пользователем размеров окна. А в списке **ListBox1** при увеличении размера даже исчезла полоса прокрутки, т.к. весь текст уместился на экране.

Для того, чтобы это нормально работало, надо еще в обработчик события **OnResize** формы или панели, содержащей метку **StaticText1**, вставить оператор

```
StaticText1->Repaint();
```

Рис. 4.8

Изменение размеров компонентов при использовании привязки **Anchors**



Этот оператор обращается к методу **Repaint**, который перерисовывает метку. Если этого не сделать, то, как вы легко можете проверить, при изменении размеров окна в метке появится надпись на новом месте, но не исчезнет и прежняя надпись, т.е. получится двойная надпись, что, конечно, недопустимо.

В приведенном примере неоправданно изменился размер метки **StaticText1**. Такое поведение компонента не всегда желательно. В подобных случаях надо переходить к программному изменению местоположения компонента.

Местоположение компонента задается свойствами **Left** и **Top**, характеризующими координаты левого верхнего угла компонента. При этом началом координат считается левый верхний угол клиентской области компонента-контейнера. Горизонтальная координата отсчитывается вправо от этой точки, а вертикальная — вниз.

Размеры компонента определяются свойствами **Width** — ширина и **Height** — высота. Есть еще одно свойство **BoundsRect**, характеризующее одновременно и местоположение, и размеры компонента, но оно менее удобно и используется редко.

Имеется еще два свойства, определяющие размер клиентской области компонента-контейнера: **ClientWidth** и **ClientHeight**. Для некоторых компонентов они совпадают со свойствами **Width** и **Height**. Но могут и отличаться от них. Например, в форме **ClientHeight** меньше, чем **Height**, за счет бордюра, полосы заголовка, меню и полосы горизонтальной прокрутки, если она имеется.

Поскольку нам надо изменять местоположение и размеры компонентов при изменении размеров формы, то соответствующие операторы следует размещать в обработчиках событий **OnResize** формы или соответствующей панели. Эти события возникают при любом изменении пользователем размеров окна приложения или размеров панели. В нашем примере оператор обработки этих событий может иметь вид:

```
StaticText1->Left = Panel2->Left +  
    (Panel2->ClientWidth - StaticText1->Width) / 2;
```

Этот оператор сдвигает левый край компонента **StaticText1** так, чтобы метка всегда размещалась в середине панели **Panel2**. Только чтобы это нормально работало, над убрать привязку метки к правому краю панели, которая была введена нами ранее. Иначе размер метки все-таки будет меняться.

4.2.3 Панели с перестраиваемыми границами

В ряде случаев описанного автоматического изменения размеров различных панелей оказывается недостаточно для создания удобного пространства для действий пользователя. Даже при увеличении окна на весь экран какая-то панель приложения может оказаться перегруженной информацией, а другая относительно пустой. В этих случаях полезно предоставить пользователю возможность перемещать границы, разделяющие различные панели, изменяя их относительные размеры. Пример такой возможности можно увидеть в программе Windows «Проводник».

В библиотеке C++Builder 5 на странице Additional имеется специальный компонент — **Splitter**, который позволяет легко осуществить это. При работе с ним надо соблюдать определенную последовательность проектирования. Если вы хотите установить **Splitter** между двумя панелями, первая из которых будет выровнена к какому-то краю клиентской области, а вторая займет всю клиентскую область, то сначала надо выровнять первую панель, например, влево. Затем надо перенести на форму **Splitter** и выровнять его в ту же сторону (тоже влево). **Splitter** прижмется к соответствующему краю первой панели. После этого можно задать выравнивание второй панели на всю оставшуюся площадь клиентской области. В результате **Splitter** окажется зажатым между двумя панелями и при запуске приложения он

позволит пользователю изменять положение соответствующей границы между этими панелями.

Подобные разделители **Splitter** можно разместить между всеми панелями приложения, дав пользователю полную свободу изменять топологию окна, с которым он работает.

Компонент **Splitter** имеет событие **OnMoved**, которое наступает после конца перемещения границы. В обработчике этого события надо предусмотреть, если необходимо, упорядочение размещения компонентов на панелях, размеры которых изменились: переместить какие-то метки, изменить размеры компонентов и т.д.

Свойство **ResizeStyle** компонента **Splitter** определяет поведение разделителя при перемещении его пользователем. Поэкспериментируйте с ним, чтобы увидеть различие в режимах перемещения разделителя. По умолчанию свойство **Splitter** равно **rsPattern**. Это означает, что пока пользователь тянет курсором мыши границу, сам разделитель не перемещается и панели тоже остаются прежних размеров. Перемещается только шаблон линии, указывая место намечаемого перемещения границы. Лишь после того, как пользователь отпустит кнопку мыши, разделитель переместится и панели изменят свои размеры. Практически такая же картина наблюдается, если установить **ResizeStyle = rsLine**. При **ResizeStyle = rsUpdate** в процессе перетаскивания границы пользователем разделитель тоже перемещается и размеры панелей все время меняются. Это, может быть, удобно, если пользователь хочет установить размер панели таким, чтобы на ней была видна какая-то конкретная область. Но так как процесс перетаскивания в этом случае сопровождается постоянной перерисовкой панелей, наблюдается неприятное мерцание изображения. Так что этот режим можно рекомендовать только в очень редких случаях. Если установить **ResizeStyle = rsNone**, то в процессе перетаскивания границы не перемещается ни сама граница, ни изображающая ее линия. Вряд ли это удобно пользователю, так что я не могу представить случая, когда было бы выгодно использовать этот режим.

Свойство **MinSize** компонента **Splitter** устанавливает минимальный размер в пикселях обеих панелей, между которыми зажат разделитель. Задание такого минимального размера необходимо, чтобы при перемещениях границы панель не сжалась бы до нулевого размера или до такой величины, при которой на ней исчезли бы какие-то необходимые для работы элементы управления. К сожалению, в версиях C++Builder, младше C++Builder 5, свойство **MinSize** не всегда срабатывает верно. В C++Builder 5 введено новое свойство компонента **Splitter** — **AutoSnap**. Если оно установлено в **true** (по умолчанию), то при перемещении границы возможны те же неприятности, что в младших версиях C++Builder. Но если установить **AutoSnap** в **true**, то перемещение границы панелей сверх пределов, при которых размер одной из панелей станет меньше **MinSize**, просто блокируется. Так что можно рекомендовать всегда устанавливать **AutoSnap** в **true**.

Впрочем, и это не решает всех задач, связанных с перемещением границ панелей. Дело в том, что свойство **MinSize** относится к обеим панелям, граница между которыми перемещается, а в ряде случаев желательно отдельно установить различные минимальные размеры одной и другой панели. Это проще сделать, задав в панелях соответствующие значения свойства **Constraints**, о котором будет рассказано в следующем разделе.

4.2.4 Ограничение пределов изменения размеров окон и компонентов

Все рассмотренные ранее методы изменения размеров панелей и компонентов на них имеют общий недостаток: при чрезмерном уменьшении пользователем размеров окна какие-то компоненты могут исчезать из поля зрения. Иногда к некрассивым с точки зрения эстетики результатам приводит и чрезмерное увеличение

размеров окна. Хотелось бы иметь средства, ограничивающие пользователя в его манипуляциях с окном и не позволяющие ему чрезмерно уменьшать и увеличивать размеры.

Таим средством является свойство **Constraints**, присущее всем компонентам и позволяющее задавать ограничения на допустимые изменения размеров. Свойство имеет четыре основных подсвойства: **MaxHeight**, **MaxWidth**, **MinHeight** и **MinWidth** — соответственно максимальная высота и ширина и минимальная высота и ширина. По умолчанию значения всех этих подсвойств равны 0, что означает отсутствие ограничений. Но задание любому из этих свойств положительного значения приводит к соответствующему ограничению размера заданным числом пикселей.

Чтобы какие-то компоненты не исчезали из поля зрения, можно задать им ограничения минимальной высоты и длины. Таким образом можно поддерживать нормальные пропорции отдельных частей окна. Можно задать ограничения на минимальные и максимальные размеры формы, т.е. всего окна. Например, если вы зададите для формы значения **MaxHeight** = 500 и **MaxWidth** = 500, то пользователь не сможет сделать окно большим, чем квадрат 500 x 500. Причем это ограничение будет действовать даже если пользователь нажмет системную кнопку, разворачивающую окно на весь экран. Окно развернется, но его размеры не превысят заданных. Это иногда полезно делать, чтобы развернутое окно не заслонило какие-то другие нужные пользователю окна.

4.2.5 Масштабирование компонентов

Говоря о размерах компонентов следует упомянуть еще об одной возможности их изменения — о методе **ScaleBy**. Он определен как:

```
void __fastcall ScaleBy(int M, int D);
```

где **M** и **D** — множитель и делитель, определяющие изменение масштаба компонента. Масштабируются такие свойства компонента, как **Width** и **Height**, определяющие его размер. Свойства **Top** и **Left** остаются неизменными. Масштабируется также размер шрифта, если только в компоненте не установлено **ParentFon** = **true**. В последнем случае шрифт наследуется от родительского компонента и поэтому не изменяется.

Если компонент является контейнером, содержащим другие компоненты, то его дочерние компоненты также масштабируются. Причем у них изменяются не только **Width** и **Height**, но также пропорционально изменяются **Top** и **Left**, определяющие их местоположение. Если во всех дочерних компонентах установлено **ParentFont** = **true**, а в компоненте-контейнере **ParentFont** = **false**, то пропорционально изменяются и шрифты всех компонентов (но, конечно, не непрерывно, а скачками, доступными тому или иному типу шрифта).

Параметры **M** и **D** определяют соответственно множитель и делитель масштаба. Например, чтобы уменьшить размеры на 10% от начального значения, можно задать **M** равным 9, а **D** равным 10 (9/10). Если же вы хотите увеличить размер на 1/3, то можно задать **M**=133 и **D**=100 (133/100) или **M**=4 и **D**=3 (4/3).

Приведем примеры. Оператор

```
Edit1->ScaleBy(11,10);
```

масштабирует окно редактирования **Edit1**. В любом случае при выполнении этого оператора длина окна (свойство **Width**) увеличивается на 10%, что обеспечивает возможность наблюдать и редактировать в нем более длинный текст. Высота окна (свойство **Height**) будет изменяться пропорционально, только если свойство компонента **AutoSize** равно **false**. В противном случае высота определяется только размером шрифта и при постоянном шрифте будет неизменной. А размер шрифта будет меняться, только если свойство компонента **ParentFont** равно **false**, т.к. ина-

че шрифт определяется родительским компонентом. Таким образом при **AutoSize = true** и **ParentFont = true** изменяется только длина окна редактирования.

Оператор

```
Panel1->ScaleBy(11,10);
```

увеличивает размер панели, а также координаты и размер всех ее компонентов. Если в панели **ParentFont = false**, то шрифты во всех компонентах также увеличиваются независимо от значений их свойства **ParentFont**. Причем они увеличиваются и в неоконных компонентах, например, в метках. Если же в панели **ParentFont = true**, то шрифты увеличиваются только в компонентах, в которых **ParentFont = false**.

Приведенный выше оператор изменяет масштаб сразу группы компонентов, но при этом сдвигает их позиции, поскольку действует на свойства **Top** и **Left**. Можно избежать этого, обращаясь по отдельности к каждому дочернему оконному компоненту панели с помощью, например, такого оператора:

```
for (int i = 0; i < Panel1->ControlCount; i++)
    if (Panel1->Controls[i]->InheritsFrom(__classid(TWinControl)))
        ((TWinControl *)Panel1->Controls[i])->ScaleBy(11,10);
```

При этом надо иметь в виду, что непосредственно применять **ScaleBy** можно только к оконным компонентам, являющимся наследниками класса **TWinControl**. Поэтому в приведенном операторе сначала проверяется, является ли компонент наследником **TWinControl**. Для этого к компоненту применяется функция **InheritsFrom**, возвращающая **true**, если класс объекта наследует классу, передаваемому в функцию как параметр. В данном случае в качестве параметра в функцию передается класс **TWinControl**, который преобразуется в нужную форму с помощью ключевого слова **__classid**.

Если установлено, что компонент является оконным (его класс наследует **TWinControl**), то к нему применяется функция **ScaleBy**, для чего используется явное приведение его типа к **TWinControl**: **(TWinControl *)**.

Приведенный код масштабирует только оконные компоненты. Неоконные компоненты, например, метки, масштабироваться не будут.

Подробнее функции и подходы, использованные в приведенном коде, рассмотрены в главе 1 в разделе 1.6.6.2.

4.3 Обработка событий клавиатуры и мыши

Все действия пользователя при взаимодействии с приложением сводятся к перемещению мыши, нажатию кнопок мыши и нажатию клавиш клавиатуры. Рассмотрим обработку в приложении событий, связанных с этими манипуляциями пользователя.

4.3.1 События мыши

4.3.1.1 Последовательность событий

В компонентах C++Builder определен ряд событий, связанных с мышью. Это события:

Событие	Описание
OnClick	Щелчок мыши на компоненте и некоторые другие действия пользователя.
OnDblClick	Двойной щелчок мыши на компоненте.

OnMouseDown	Нажатие клавиши мыши над компонентом. Возможно распознавание нажатой кнопки и координат курсора мыши.
OnMouseMove	Перемещение курсора мыши над компонентом. Возможно распознавание нажатой кнопки и координат курсора мыши.
OnMouseUp	Отпускание ранее нажатой кнопки мыши над компонентом. Возможно распознавание нажатой кнопки и координат курсора мыши.
OnStartDrag	Начало процесса «перетаскивания» объекта. Возможно распознавание перетаскиваемого объекта.
OnDragOver	Перемещение «перетаскиваемого» объекта над компонентом. Возможно распознавание перетаскиваемого объекта и координат курсора мыши.
OnDragDrop	Отпускание ранее нажатой кнопки мыши после «перетаскивания» объекта. Возможно распознавание перетаскиваемого объекта и координат курсора мыши.
OnEndDrag	Еще одно событие при отпускании ранее нажатой кнопки мыши после «перетаскивания» объекта. Возможно распознавание перетаскиваемого объекта и координат курсора мыши.
OnEnter	Событие в момент получения элементом фокуса в результате манипуляции мышью, нажатия клавиши табуляции или программной передачи фокуса.
OnExit	Событие в момент потери элементом фокуса в результате манипуляции мышью, нажатия клавиши табуляции или программной передачи фокуса.

Как видно из приведенной таблицы, эти события охватывают все мыслимые манипуляции с мышью и даже дублируют многие из них. Наиболее широко используется событие **OnClick**. Обычно оно наступает, если пользователь щелкнул на компоненте, т.е. нажал и отпустил кнопку мыши, когда указатель мыши находился на компоненте. Но это событие происходит также и при некоторых других действиях пользователя. Оно наступает, если:

- Пользователь выбрал элемент в сетке, дереве, списке, выпадающем списке, нажав клавишу со стрелкой.
- Пользователь нажал клавишу пробела, когда кнопка или индикатор были в фокусе.
- Пользователь нажал клавишу Enter, а активная форма имеет кнопку по умолчанию, указанную свойством **Default**.
- Пользователь нажал клавишу Esc, а активная форма имеет кнопку прерывания, указанную свойством **Cancel**.
- Пользователь нажал клавиши быстрого доступа к кнопке или индикатору. Например, если свойство **Caption** индикатора записано как «&Полужирный» и символ 'П' подчеркнут, то нажатие пользователем комбинации клавиш Alt+П вызовет событие **OnClick** в этом индикаторе.
- Приложение установило в **true** свойство **Checked** радиокнопки **RadioButton**.
- Приложение изменило свойство **Checked** индикатора **CheckBox**.
- Вызван метод **Click** элемента меню.

Для формы событие **OnClick** наступает, если пользователь щелкнул на пустом месте формы или на недоступном компоненте.

Обилие событий, связанных с мышью, а также фактическое дублирование некоторых из них требуют четкого представления о последовательности отдельных событий, наступающих при том или ином действии пользователя. Рассмотрим эти последовательности.

В момент запуска приложения из рассматриваемых нами событий наступает только событие **OnEnter** в компоненте, на который передается фокус. Это событие не связано с какими-то действиями пользователя, так что не будем на нем останавливаться.

Теперь рассмотрим простейшее действие пользователя: переключение с помощью мыши фокуса с одного элемента на другой. Последовательность событий в этом случае приведена в таблице 4.1.

Таблица 4.1. Последовательность событий мыши при переключении фокуса

Действие пользователя	Событие
Перемещение курсора мыши в пределах первого компонента	Множество событий OnMouseMove в первом компоненте
Перемещение курсора мыши в пределах формы	Множество событий OnMouseMove в форме
Перемещение курсора мыши в пределах второго компонента	Множество событий OnMouseMove во втором компоненте
Нажатие кнопки мыши	OnExit в первом компоненте OnEnter во втором компоненте OnMouseDown во втором компоненте
Отпускание кнопки мыши	OnClick во втором компоненте OnMouseUp во втором компоненте

События **OnMouseMove** происходят постоянно в процессе перемещения курсора мыши и даже просто при его дрожании, неизбежном, если пользователь не снимает руки с мыши. Это надо учитывать и пользоваться этим событием очень осторожно, поскольку оно, в отличие от других, происходит многократно.

Как видно из приведенной таблицы, каждое действие пользователя, связанное с нажатием или отпусканием кнопки мыши приводит к серии последовательно наступающих событий. В обработчиках событий **OnMouseDown** и **OnMouseUp** можно распознать, какая кнопка мыши нажата и в какой точке компонента находится в данный момент курсор мыши.

Рассмотренная в таблице 4.1 последовательность событий имеет место, если во втором компоненте свойство **DragMode** (см. раздел 4.4) равно **dmManual** (ручное начало процесса перетаскивания), как это установлено по умолчанию. Если же это свойство равно **dmAutomatic** (автоматическое начало процесса перетаскивания), то все рассмотренные события, связанные с манипуляцией мышью, заменяются следующими:

OnMouseDown	Заменяется на OnStartDrag
OnMouseMove	Заменяется на событие OnDragOver того компонента, над которым перемещается курсор мыши
OnMouseUp	Заменяется на событие OnDragDrop компонента, над которым завешается перетаскивание (если компонент может воспринять информацию от перетаскиваемого объекта), и последующее событие OnEndDrag компонента, который перетаскивался

События **OnExit** и **OnEnter** вообще не возникают, поскольку переключения фокуса не происходит. Не наступает также событие **OnClick**.

В дальнейшем в данном разделе события, связанные с перетаскиванием, рассматриваться не будут. Подробное описание этих событий см. в разделе 4.4.

Если в примере, приведенном в таблице 4.1, щелчок делается на объекте, который уже находится в этот момент в фокусе, то не происходят события **OnExit** и **OnEnter**. В этом случае при нажатии кнопки наступает только событие **OnMouseDown**, а при отпускании кнопки — события **OnClick** и **OnMouseUp**.

Теперь рассмотрим последовательность событий при двойном щелчке на компоненте. Она приведена в таблице 4.2. Распознавать нажатую кнопку мыши по-прежнему можно только в событиях **OnMouseDown** и **OnMouseUp**. Если же надо распознать именно двойной щелчок какой-то определенной кнопкой мыши, то можно, например, ввести некую переменную, являющуюся флагом двойного щелчка, устанавливать этот флаг в обработчике события **OnDbClick**, а в обработчиках событий **OnMouseDown** или **OnMouseUp** проверять этот флаг и, если он установлен, то сбрасывать его и выполнять запланированные действия.

Таблица 4.2. Последовательность событий мыши при двойном щелчке на компоненте

Действие пользователя	Событие
Первое нажатие кнопки мыши	OnMouseDown . Возможно распознавание нажатой кнопки и координат курсора мыши
Первое отпускание кнопки мыши	OnClick OnMouseUp . Возможно распознавание нажатой кнопки и координат курсора мыши
Второе нажатие кнопки мыши	OnDbClick OnMouseDown . Возможно распознавание нажатой кнопки и координат курсора мыши
Второе отпускание кнопки мыши	OnMouseUp . Возможно распознавание нажатой кнопки и координат курсора мыши.

4.3.1.2 Распознавание источника события, нажатых кнопок и клавиш, координат курсора

Во все обработчики событий, связанных с манипуляциями мыши (как и во все другие обработчики) передается параметр **Sender** типа указателя на **TObject**. Этот параметр содержит указатель на компонент, в котором произошло событие. Он не требуется, если пишется обработчик события для одного конкретного компонента. Однако часто один обработчик применяется для нескольких компонентов. При этом какие-то операции могут быть общими для любых источников события, а какие-то требовать специфических действий. Тогда **Sender** можно использовать для распознавания источника события. Правда, поскольку тип **TObject** не имеет никаких полезных для пользователя свойств и методов, то объект **Sender** следует рассматривать как объект одного из производных от **TObject** типов.

Операции, связанные с обработкой параметра **Sender**, подробно рассмотрены в главе 1 в разделе 1.6.6.2. Там показано, как можно распознать объект, на который указывает **Sender**, по имени, узнать его класс, определить, является ли этот класс наследником какого-то другого определенного класса.

Помимо параметра **Sender** в обработчики событий **OnMouseDown** и **OnMouseUp** передаются параметры, позволяющие распознать нажатую кнопку, нажатые

при этом вспомогательные клавиши, а также определить координаты курсора мыши. Заголовок обработчика события **OnMouseDown** или **OnMouseUp** может иметь, например, следующий вид:

```
void __fastcall TForm1::Edit1MouseDown(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
```

Помимо уже рассмотренного нами параметра **Sender** в обработчик передаются параметры **Button**, **Shift**, **X** и **Y**.

Параметр **Button** типа **TMouseButton** определяет нажатую в этот момент кнопку мыши. Тип **TMouseButton** — перечислимый тип, определяемый следующим образом:

```
enum TMouseButton { mbLeft, mbRight, mbMiddle };
```

Значение **mbLeft** соответствует нажатию левой кнопки мыши, значение **mbRight** — правой, а значение **mbMiddle** — средней. Например, если вы хотите, чтобы обработчик реагировал на нажатие только левой кнопки, вы можете его первым оператором написать:

```
if (Button != mbLeft) return;
```

Тогда, если значение **Button** не равно **mbLeft**, т.е. нажата не левая кнопка, выполнение обработчика прервется.

Параметр **Shift** типа **TShiftState** определяет, какие вспомогательные клавиши на клавиатуре нажаты в момент нажатия кнопки мыши. Тип **TShiftState** — множество, определенное следующим образом:

```
enum Classes_1 { ssShift, ssAlt, ssCtrl, ssLeft, ssRight,
    ssMiddle, ssDouble };
typedef Set<Classes_1, ssShift, ssDouble> TShiftState;
```

Элементы этого множества соответствуют нажатию клавиш **Shift** (**ssShift**), **Alt** (**ssAlt**), **Ctrl** (**ssCtrl**), а также кнопок: левой (**ssLeft**), правой (**ssRight**), средней (**ssMiddle**). Информация о нажатых кнопках в параметрах **Button** и **Shift** имеет различный смысл. Параметр **Button** соответствует кнопке, нажимаемой в данный момент, а параметр **Shift** содержит информацию о том, какие кнопки были нажаты, включая и те, которые были нажаты ранее. Например, если пользователь нажмет левую кнопку мыши, а затем, не отпуская ее, нажмет правую, то после первого нажатия множество **Shift** будет равно [**ssLeft**], а после второго — [**ssLeft**, **ssRight**]. Если до этого пользователь нажал и не отпустил какие-то вспомогательные клавиши, то информация о них также будет присутствовать в множестве **Shift**.

Поскольку **Shift** является множеством, проверять наличие в нем тех или иных элементов надо методом **Contains** (см. раздел 13.6 в главе 13). Например, если вы хотите прореагировать на событие, заключающееся в нажатии левой кнопки мыши при нажатой клавише **Alt**, можно использовать оператор:

```
if ((Button == mbLeft) && (Shift.Contains(ssAlt)))
    ...
```

В приведенной структуре **if** первое условие (**Button == mbLeft**) можно заменить эквивалентным ему условием, проверяющим параметр **Shift**:

```
if (Shift.Contains(ssLeft) && (Shift.Contains(ssAlt)))
    ...
```

Аналогичные параметры **Button** и **Shift** передаются и в обработчик события **OnMouseUp**. Отличие только в том, что параметр **Button** соответствует не нажимаемой в данный момент, а отпускаемой кнопке. Параметр **Shift** передается также в обработчик события **OnMouseMove**, так что и в этом обработчике можно определить, какие клавиши и кнопки нажаты.

Во все события, связанные с мышью, передаются также координаты курсора *X* и *Y*. Эти параметры определяют координаты курсора в клиентской области компонента. Благодаря этому можно обеспечить различную реакцию в зависимости от того, в какой части клиентской области расположен курсор.

4.3.2 События клавиатуры

4.3.2.1 Последовательность событий

В оконных компонентах C++Builder определены три события, связанные с клавиатурой. Это события:

Событие	Описание
OnKeyDown	Событие наступает при нажатии пользователем любой клавиши. Можно распознать нажатые клавиши, включая функциональные, и кнопки мыши, но нельзя распознать символ нажатой клавиши
OnKeyPress	Событие наступает при нажатии пользователем клавиши символа. Можно распознать только нажатую клавишу символа, различить символ в верхнем и нижнем регистре, различить символы кириллицы и латинские, но нельзя распознать функциональные клавиши и кнопки
OnKeyUp	Событие наступает при отпускании пользователем любой клавиши. Можно распознать нажатые клавиши, включая функциональные, и кнопки мыши, но нельзя распознать символ отпускаемой клавиши

Кроме того, при нажатии пользователем клавиши табуляции фокус может переключаться с элемента на элемент, что вызывает описанные в разделе 4.3.1.1 события **OnEnter** и **OnExit**.

Важно четко представлять последовательность событий, происходящих при нажатии пользователем клавиши или комбинации клавиш. Пусть, например, пользователь нажал клавишу Shift (перевел ввод в верхний регистр), а затем нажал клавишу символа 'н'. Последовательность событий для этого случая приведена в таблице 4.3. В таблице указано, что именно можно распознать при каждом событии. Подробнее это будет рассмотрено ниже, а пока отметим, что различить символ в верхнем и нижнем регистрах и различить латинский символ и символ кириллицы можно только в обработчике события **OnKeyPress**. Действительно, хотя в событии **OnKeyDown** при нажатии клавиши 'н' можно определить, что при этом одновременно нажата и клавиша Shift, этого еще мало, чтобы утверждать, что символ относится к верхнему регистру. Ведь если перед этим была включена клавиша CapsLock, то при нажатой клавише Shift символ окажется в нижнем регистре. А информация о том, включена или выключена клавиша Capslock, в обработчик события **OnKeyDown** не передается.

Таблица 4.3. Последовательность событий клавиатуры при нажатии клавиш Shift-н

Действие пользователя	Событие
Нажатие клавиши Shift	OnKeyDown. Возможно распознавание нажатой клавиши Shift
Нажатие клавиши «н»	OnKeyDown. Возможно распознавание нажатой клавиши Shift, нажатой клавиши «н», но отличить верхний регистр от нижнего и латинский символ от русского невозможно
	OnKeyPress. Возможно распознавание символа с учетом регистра и языка, но невозможно распознавание нажатой клавиши Shift
Отпускание клавиши «н»	OnKeyUp. Возможно распознавание нажатой клавиши Shift, отпущенной клавиши «н», но отличить верхний регистр от нижнего и латинский символ от русского невозможно
Отпускание клавиши Shift	OnKeyUp. Возможно распознавание отпущенной клавиши Shift

Следует отметить, что событие **OnKeyPress** заведомо наступает, если нажимается только клавиша символа или клавиша символа при нажатой клавише Shift. Если же клавиша символа нажимается одновременно с какой-то из вспомогательных клавиш, то событие **OnKeyPress** может не наступить (произойдут только события **OnKeyDown** при нажатии и **OnKeyUp** при отпускании) или, если и наступит, то укажет на неверный символ. Например, при нажатой клавише Alt событие **OnKeyPress** при нажатии символьной клавиши не наступает. А при нажатой клавише Ctrl событие **OnKeyPress** при нажатии символьной клавиши наступает, но символ не распознается.

В заключение надо остановиться на вопросе, куда поступают события клавиатуры. У формы имеется свойство **KeyPreview**. Оно влияет на обработку событий, поступающих от клавиатуры (в число этих событий не входит нажатие клавиш со стрелками, клавиш табуляции и т.п.). По умолчанию свойство **KeyPreview** равно **false** и события клавиатуры поступают на обработчики, предусмотренные в активном в данный момент компоненте. Но если задать значение **KeyPreview** равным **true**, то сначала эти события будут поступать на обработчики формы, если таковые предусмотрены, и только потом поступят на обработчики активного компонента.

Имеется также событие **OnShortCut** приложения (**Application**), которое возникает при нажатии пользователем клавиши. Событие возникает до того, как возникло стандартное событие **OnKeyDown** компонента или формы. Это событие, как и все события приложения, перехватывает введенный в C++Builder 5 компонент **ApplicationEvents**. Обработчик этого события позволяет предусмотреть нестандартную реакцию на нажатие какой-то клавиши. В него передается параметр сообщения Windows **Msg**, поле **CharCode** которого (**Msg.CharCode**) содержит виртуальный код нажатой клавиши. Передается также по ссылке параметр **Handled**. Если задать ему значение **true**, то стандартные события **OnKeyDown**, **OnKeyPress**, **OnKeyUp** не наступят. В разделе 3.9.3 приведен пример использования обработчика события **OnShortCut**.

4.3.2.2 Распознавание нажатых клавиш

Заголовок обработчика события **OnKeyDown** может иметь, например, следующий вид:

```
void __fastcall TForm1::Edit1KeyDown(TObject *Sender,
                                     WORD &Key, TShiftState Shift)
```

Параметр **Sender**, указывающий на источник события, уже обсуждался выше при описании событий мыши (см. раздел 4.3.1.2). Там же рассматривался и параметр **Shift**, представляющий собой множество элементов, отражающих нажатые в это время функциональные клавиши. Только в обработчике события **OnKeyDown** множество возможных элементов параметра **Shift** сокращено до **ssShift** (нажата клавиша Shift), **ssAlt** (нажата клавиша Alt) и **ssCtrl** (нажата клавиша Ctrl). Информация о нажатых кнопках мыши отсутствует.

Основной параметр, которого не было раньше — это параметр **Key**. Обратите внимание, что он передается по ссылке, т.е. может изменяться в обработчике события. Кроме того, обратите внимание, что это целое число, а не символ.

Параметр **Key** определяет нажатую в момент события клавишу клавиатуры. Для не алфавитно-цифровых клавиш используются виртуальные коды API Windows. Полная таблица этих кодов приведена в справочной части книги в главе 15 в разделе 15.1.1. Ниже в таблице 4.4 приведены для дальнейшего обсуждения только несколько строк из нее, соответствующих наиболее распространенным клавишам.

Таблица 4.4. Некоторые коды клавиш

Клавиша	Десятичное число	Шестнадцатеричное число	Символическое имя	Сравнение по символу
F1	112	0x70	VK_F1	
Enter	13	0x0D	VK_RETURN	
Shift	16	0x10	VK_SHIFT	
Ctrl	17	0x11	VK_CONTROL	
Alt	18	0x12	VK_MENU	
Esc	27	0x1B	VK_ESCAPE	
0)	48	0x30		0
1 !	49	0x31		1
n N т Т	78	0x4E		N
y Y н Н	89	0x59		Y

Параметр **Key** является целым числом, определяющим клавишу, а не символ. Например, один и тот же код соответствует прописному и строчному символам «Y» и «y». Если, как это обычно бывает, в русской клавиатуре этой клавише соответствуют символы кириллицы «Н» и «н», то их код будет тем же самым. Различить прописные и строчные символы или символы латинские и кириллицы невозможно.

Проверить нажатую клавишу можно, сравнивая **Key** с целым десятичным кодом клавиши, приведенном во втором столбце таблицы 4.4. Например, реакцию на нажатие пользователем клавиши Enter можно оформить оператором:

```
if (Key == 13) ... ;
```

Можно сравнивать **Key** и с шестнадцатеричным эквивалентом кода, приведенным в третьем столбце таблицы 4.4. Например, приведенный выше оператор можно записать в виде:

```
if(Key == 0x0D) ... ;
```

Для клавиш, которым не соответствуют символы, введены также именованные константы, которые облегчают написание программы, поскольку не требуют помнить численные коды клавиш. Например, приведенный выше оператор можно записать в виде:

```
if(Key == VK_RETURN) ... ;
```

Для клавиш символов и цифр можно производить проверку сравнением с десятичным или шестнадцатеричным кодом, но это не очень удобно, так как трудно помнить коды различных символов. Другой путь — воспользоваться тем, что коды латинских символов в верхнем регистре совпадают с виртуальными кодами, используемыми в параметре **Key**. Поэтому, например, если вы хотите распознать клавишу, соответствующую символу «Y», вы можете написать:

```
if(Key == 'Y') ... ;
```

В этом операторе можно использовать только латинские символы в верхнем регистре. Если вы напишете 'y' или захотите написать русские символы, соответствующие этой клавише — 'Н' или 'н', то оператор не работает.

Помните также, что оператор будет действовать на все символы, относящиеся к указанной клавише: «Y», «y», «H» и «h». Иногда это хорошо, а иногда плохо. Например, если вы задали пользователю какой-то вопрос, на который он должен ответить Y (да) или N (нет), то подобный оператор избавляет пользователя от необходимости следить, в каком регистре он вводит символ и какой язык — английский или русский включен в данный момент. Ему достаточно просто нажать клавишу, на которой написано «Y». Однако, если пользователь более привык к символам кириллицы, то могут возникнуть неприятности, поскольку нажимая клавишу с латинской буквой «Y» и русской буквой «Н» он может думать, что отвечает не положительно (Yes — да), а отрицательно (Нет).

Приведем еще один пример — автоматическую передачу фокуса очередному оконному компоненту при нажатии пользователем клавиши Enter. Это можно сделать, включив в общий обработчик событий **OnKeyDown** всех оконных компонентов оператор:

```
if(Key == VK_RETURN)
    FindNextControl((TwinControl *)Sender, true, true,
                    false)->SetFocus();
```

Этот оператор с помощью метода **FindNextControl** ищет очередной компонент в последовательности табуляции и передает ему фокус. Подробнее этот пример и метод **FindNextControl** рассмотрен в разделе 4.1.8.

В заключение приведем пример распознавания комбинации клавиш. Пусть вы хотите, например, распознать комбинацию Alt-X. Для этого вы можете написать оператор:

```
if((Key == 'X') && Shift.Contains(ssAlt)) ... ;
```

Мы рассмотрели распознавание клавиш при событии **OnKeyDown**. Заголовок обработчика события **OnKeyUp** имеет такой же вид, так что все сказанное в равной степени относится и к событиям при отпуске клавиш.

Теперь перейдем к рассмотрению события **OnKeyPress**. Заголовок обработчика этого события имеет вид:

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender,
                                     char &Key)
```

В этот обработчик, как и в описанные выше, передается параметр **Key**, определяющий нажатую клавишу символа. Но обратите внимание, что тип этого параметра не целое число, как в предыдущих случаях, а **char** — символ. В данном случае в обработчик передается не виртуальный код клавиши, а символ, по которому можно определить, прописная это буква, или строчная, русская, или латинская. Поэтому описанных выше сложностей с распознаванием символов не возникает.

Пусть, например, вы задали пользователю вопрос, на который он должен ответить символами «Д» или «д» (да), или символами «Н» или «н» (нет). Тогда распознать положительный ответ в обработчике события **OnKeyPress** можно оператором:

```
if ((Key == 'Д') || (Key == 'д'))...
```

Приведенный оператор реагирует только на положительный ответ пользователя, не реагируя на отрицательный или ошибочный ответ. Реакцию на все возможные ответы обеспечивает структура множественного выбора **switch** (см. раздел 12.8.1.2 в главе 12):

```
switch (Key)
{
    case 'Д':
    case 'д': ...;
                break;
    case 'Н':
    case 'н': ...;
                break;
    default: Beep();
}
```

Здесь предусмотрена реакция на положительный и отрицательный ответ, а также звуковой сигнал при ошибочном ответе.

Посмотрев на приведенный ранее заголовок обработчика, вы можете увидеть, что параметр **Key** передается по ссылке. Это позволяет в обработчике изменять **Key**, изменяя соответственно его стандартную обработку в компоненте, поскольку ваш обработчик события срабатывает раньше стандартного обработчика компонента. Пусть, например, вы хотите иметь на форме окно редактирования **Edit1**, в котором пользователь должен вводить только целые числа без знака. Вы можете обеспечить безошибочный ввод, подменяя все недопустимые символы нулевым с помощью, например, следующего кода:

```
Set <char, '0', '9'> Dig;
Dig << '0' << '1' << '2' << '3' << '4' << '5'
    << '6' << '7' << '8' << '9';
if ( ! Dig.Contains(Key))
{ Key = 0; Beep();}
```

При нажатии пользователем любой клавиши, кроме клавиш с цифрами, символы подменяются нулевым символом и просто не появляются в окне редактирования, как вы можете убедиться, сделав приложение с этим простым примером. Функция **Вееп** воспроизводит при нажатии пользователем ошибочной клавиши звуковой сигнал.

4.4 Перетаскивание объектов

4.4.1 Перетаскивание информации об объектах — технология Drag&Drop

Процесс перетаскивания с помощью мыши информации из одного объекта в другой (**Drag&Drop**), коротко называемый перетаскиванием, очень широко используется в Windows. Например, вы можете перемещать файлы между папками,

перемещать сами папки, включая их в другие папки, и т.д. Посмотрим, как осуществляется подобное перетаскивание информации об объектах в C++Builder.

Все свойства, методы и события, связанные с процессом перетаскивания, определены в классе **TControl**, являющемся прародителем всех визуальных компонентов C++Builder. Поэтому они являются общими для всех компонентов.



Начало процесса перетаскивания определяется свойством **DragMode**, которое может устанавливаться в процессе проектирования или программно равным **dmManual** или **dmAutomatic**. Значение **dmAutomatic** (автоматическое) определяет автоматическое начало процесса перетаскивания при нажатии пользователем кнопки мыши над компонентом. Имейте в виду, что в этом случае событие **OnMouseDown**, связанное с нажатием пользователем кнопки мыши, для этого компонента вообще не наступает. Значение же **dmManual** (ручное) говорит о том, что начало процесса перетаскивания должен определять программист. Для этого он должен в соответствующий момент вызвать метод **BeginDrag**. Например, он может поместить вызов этой функции в обработчик события **OnMouseDown**, наступающего в момент нажатия кнопки мыши. В этом обработчике он может проверить предварительно какие-то условия (режим работы приложения, нажатие тех или иных кнопок мыши и вспомогательных клавиш) и при выполнении этих условий вызвать **BeginDrag**.

Пусть, например, процесс перетаскивания должен начаться, если пользователь нажал левую кнопку мыши и клавишу **Alt** над списком **ListBox1**. Тогда свойство **DragMode** этого компонента надо установить в **dmManual**, а его обработчик события **OnMouseDown** может иметь вид:

```
void __fastcall TForm1::ListBox1MouseDown(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
    if ((Button == mbLeft) && Shift.Contains(ssAlt))
        ListBox1->BeginDrag(false, 5);
}
```

Параметр **Button** обработчика события **OnMouseDown** показывает, какая кнопка мыши была нажата, а параметр **Shift** является множеством, содержащим обозначения нажатых в этот момент кнопок мыши и вспомогательных клавиш клавиатуры (см. раздел 4.3.1.2). Приведенный выше оператор проверяет, нажаты ли левая кнопка мыши и клавиша **Alt**. Если нажаты, то вызывается метод **BeginDrag** данного компонента.

В предыдущем примере в функцию **BeginDrag** переданы значения параметров **false** и **5**. Первый из них означает, что процесс перетаскивания начнется не сразу, а только после того, как пользователь сдвинет мышью с нажатой при этом кнопкой на некоторое число пикселей, а второй параметр указывает величину перемещения — **5**. Это позволяет отличить простой щелчок мыши от начала перетаскивания. Если же передать в **BeginDrag** значение **true** и любое число, то перетаскивание начнется немедленно.

Когда начался процесс перетаскивания, обычный вид курсора изменяется. Пока он перемещается над формой или компонентами, которые не могут принять информацию, он обычно имеет тип **crNoDrop**: . Если же он перемещается над компонентом, готовым принять информацию из перетаскиваемого объекта, то приобретает вид, определяемый свойством перетаскиваемого объекта **DragCursor**. По умолчанию это свойство равно **crDrag**, что соответствует изображению . Надо подчеркнуть, что вид курсора определяется свойством **DragCursor** перетаскиваемого объекта, а не того объекта, над которым перемещается курсор.

В процессе перетаскивания компоненты, над которыми перемещается курсор, могут сообщать о готовности принять информацию от перетаскиваемого объекта. Для этого в компоненте должен быть предусмотрен обработчик события **OnDragOver**, наступающего при перемещении над данным компонентом курсора,

перетаскивающего некоторый объект. В этом обработчике надо проверить, может ли данный компонент принять информацию перетаскиваемого объекта, и если не может, задать значение **false** передаваемому в обработчик параметру **Accept**. По умолчанию этот параметр равен **true**, что означает возможность принять перетаскиваемый объект. Обработчик для списка **ListBox1** может иметь, например, следующий вид:

```
void __fastcall TForm1::ListBox1DragOver(TObject *Sender,
    TObject *Source, int X, int Y, TDragState State, bool &Accept)
{
    if(Sender != Source)
        Accept = Source->ClassNameIs("TListBox");
    else Accept = false;
}
```

В нем сначала проверяется, не являются ли данный компонент (**Sender**) и перетаскиваемый объект (**Source**) одним и тем же объектом. Это сделано, чтобы избежать перетаскивания информации внутри одного и того же списка. Если источник и приемник являются одним и тем же объектом, то срабатывает **else** и параметр **Accept** становится равным **false**, запрещая прием информации. Если же это разные объекты, то **Accept** делается равным **true**, если источником является какой-то другой список (компонент класса **TListBox**), и равным **false**, если источник является объектом любого другого типа. Таким образом компонент **ListBox1** сообщает, что готов принять информацию из любого другого списка.

Значение параметра **Accept**, задаваемое в обработчике события **OnDragOver**, определяет вид курсора, перемещающегося при перетаскивании над данным компонентом. Если в компоненте не описан обработчик события **OnDragOver**, то считается, что данный компонент не может принять информацию перетаскиваемого объекта.

Процедура приема информации от перетаскиваемого объекта записывается в обработчике события **OnDragDrop** принимающего компонента. Это событие наступает, если после перетаскивания пользователь отпустил кнопку мыши над данным компонентом. В обработчик этого события передаются параметры **Source** (объект-источник) и **X** и **Y** координаты курсора. Если продолжить начатый выше пример перетаскивания информации из одного списка в другой, то обработчик события **OnDragDrop** может иметь вид:

```
void __fastcall TForm1::ListBox1DragDrop(TObject *Sender,
    TObject *Source, int X, int Y)
{
    TListBox *S = (TListBox *)Source;
    ((TListBox*)Sender)->Items->Add(
        S->Items->Strings[S->ItemIndex]);
}
```

В этом обработчике первый оператор создает указатель **S** на объект класса **TListBox**, и передает в него ссылку на объект **Source**, воспринимаемый как объект **TListBox**. Это сделано просто для того, чтобы не повторять несколько раз в следующем операторе приведение типа **Source** к указателю на объект класса **TListBox**. А такое приведение типа необходимо по следующей причине. Параметр **Source** объявлен как указатель на объект класса **TObject**. Но в этом классе отсутствуют свойства **Items** и **ItemIndex**. Они имеются только в классе **TListBox**. Поэтому прежде, чем обратиться к этим свойствам, надо произвести соответствующее приведение типа **Source**. Подробнее это изложено в разделе 1.5.6.2 главы 1.

Второй оператор обработчика заносит методом **Add** выделенную строку списка-источника **S** в список-приемник **Sender**. В этом переносе информации из одного списка в другой и заключается в нашем примере **Drag&Drop**.

В приведенном обработчике можно было бы обойтись и без создания указателя **S**. В этом случае обработчик имел бы следующий вид:

```
void __fastcall TForm1::ListBox1DragDrop(TObject *Sender,
                                         TObject *Source, int X, int Y)
{
    ((TListBox*)Sender)->Items->Add(((TListBox *)Source)->
                                     Items->Strings[ ((TListBox *)Source)->ItemIndex]);
}
```

После завершения или прерывания перетаскивания наступает событие **OnEndDrag**, в обработчике которого можно предусмотреть какие-то дополнительные действия. Имеется также связанное с перетаскиванием событие **OnStartDrag**, которое позволяет произвести какие-то операции в начале перетаскивания. Это событие полезно при автоматическом начале перетаскивания, когда иным способом этот момент нельзя зафиксировать.

Теперь, объединив приведенные выше фрагменты обработки перетаскивания, просуммируем, что надо сделать, если вы имеете в приложении несколько списков **Listbox** и хотите обеспечить возможность копирования строк каждого из этих списков в любой другой.

Это потребует двух операций:

- Выделите мышью все списки приложения как одну группу, перейдите в Инспекторе Объектов на страницу событий, сделайте двойной щелчок около строки события **OnDragOver** и напишите в Редакторе Кода приведенный выше обработчик события **OnDragOver**.
- Аналогичным образом напишите общий для всех списков приведенный выше обработчик события **OnDragDrop**.

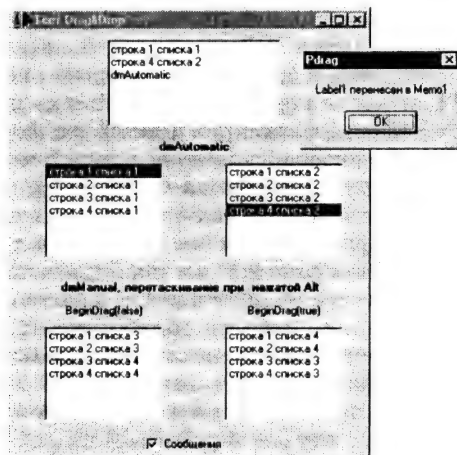
И это все! Для обеспечения возможности перетаскивания вам потребовалось написать всего два оператора. Если вы хотите начинать перетаскивание только при выполнении какого-то дополнительного условия, например, при нажатии клавиши **Alt**, то вам потребуется сделать еще один шаг:

- Задайте для всех списков значение свойства **DragMode**, равное **dmManual**. Напишите для этих списков приведенный выше общий обработчик события **OnMouseDown**.

Теперь вы можете создать форму с несколькими списками, занести в них информацию, написать эти обработчики и проверить все это на практике. На рис. 4.9 приведен пример, демонстрирующий различные аспекты перетаскивания. Этот и другие примеры имеются на диске, прилагаемом к книге. Можете попробовать для тренировки сами создать аналогичный пример, позволяющий перетаскивать строки и метки из одного списка в другой и из списков в окно редактирования **Memo**. В примере демонстрируются разные способы перетаскивания: автоматическое, руч-

Рис. 4.9

Пример приложения, использующего Drag&Drop



ное, только при определенной нажатой клавише, перетаскивание строк без удаления из списка-источника (т.е. копирование) и с удалением (перемещение). Думаю, что изложенного выше достаточно, чтоб вы сами смогли создать подобный пример.

4.4.2 Перетаскивание и встраивание объектов — Drag&Doc. Плавающие окна

Начиная с C++Builder 4 реализована технология перетаскивание и встраивание оконных объектов — Drag&Doc. Вы можете познакомиться с результатами этой технологии, работая с Интегрированной Средой Разработки C++Builder 5. Она предоставляет пользователю полную свободу в перестройке интерфейса. Отдельные окна могут объединяться вместе, создавать многостраничное окно с закладками, затем опять разъединяться и т.д. Посмотрим, как подобный подход можно реализовать в своем приложении.

У оконных компонентов введено свойство **DockSite**, которое по умолчанию равно **false**, но если его установить в **true**, то компонент становится контейнером: приемником, способным принимать переносимые в него компоненты — клиенты. Еще одно свойство приемника — **UseDockManager**. Если оно равно **true** (это принято по умолчанию), то процессом встраивания клиента автоматически управляет диспетчер встраивания, соответствующий тому компоненту, который является приемником. Если же задать **UseDockManager** равным **false**, то встраивание клиентов ложится на плечи программиста.

В компонентах — потенциальных клиентах надо установить свойство **DragKind** в **dkDock**. Кроме того, если вы хотите, чтобы процесс перетаскивания начинался автоматически, то, так же, как и в технологии Drag&Drop, надо установить свойство **DragMode** в **dmAutomatic**. Если оставить значение **DragMode** равным **dmManual** по умолчанию, то управление процессом Drag&Doc осуществляет так же, как описывалось ранее для процесса Drag&Drop.

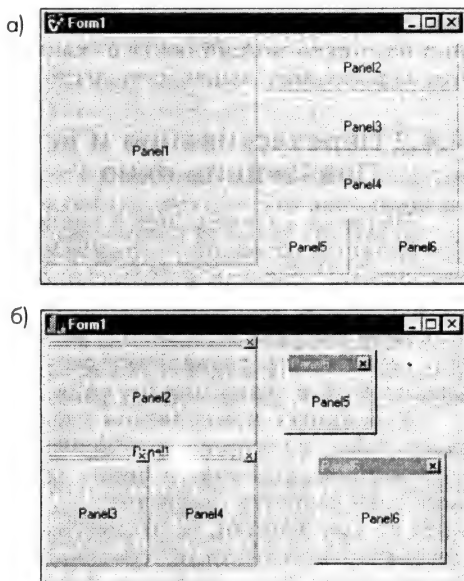
Посмотрим, к чему все это может привести. Давайте построим тестовое приложение, в котором осуществим Drag&Doc, не написав ни одной строчки кода.

Начните новое приложение. Перенесите на форму 6 панелей, расположив их примерно так, как показано на рис. 4.10 а). В панели **Panel1** установите свойство **DockSite** в **true**. Эта панель сможет служить у вас приемником. В панелях **Panel2** — **Panel6** установите **DragKind** в **dkDock** и **DragMode** в **dmAutomatic**. Запустите приложение. Попробуйте перетаскивать мышью панели **Panel2** — **Panel6**. Вы увидите (рис. 4.10 б), что панель **Panel1** может принимать другие панели, размещая их внутри себя, причем способ встраивания зависит от того, в какой части приемника вы завершаете буксировку клиента. Между размещенными клиентами имеются разделители, которые позволяют вам перемещать буксировкой границы между клиентами, изменяя их относительные размеры. Вы можете также видеть, что даже независимо от наличия или отсутствия в приложении приемника, при щелчке на компонентах-клиентах они превращаются в самостоятельные плавающие окна (панели **Panel5** и **Panel6** на рис. 4.10 б), которые можно перемещать, выходя даже за пределы родительской формы, можно изменять их размеры, можно закрывать их, после чего соответствующие компоненты исчезают. Причем все это достигнуто только заданием соответствующих свойств, без единого написанного вами оператора.

Приемником может быть и сама форма. Укажите для формы вашего приложения **DockSite** равным **true** и выполните приложение опять. Поведение перетаскиваемых панелей несколько изменится. При щелчке на компонентах-клиентах они по-прежнему превращаются в плавающие окна, которые можно перемещать, изменять их размеры и т.п. Но при повторном щелчке они опять становятся обычными панелями, сохраняя при этом измененное местоположение и размеры. Объясняет-

Рис. 4.10

Демонстрация техники Drag&Doc: форма (а) и окно приложения во время выполнения (б)



ся это тем, что повторный щелчок воспринимается в этом случае как встраивание клиента в приемник — форму.

Теперь посмотрим, как можно управлять из компонента-приемника всеми этими процессами.

У приемника имеется свойство **DockClients**, объявленное как

```
TControl* DockClients[int Index]
```

которое позволяет получить доступ к каждому клиенту, перенесенному в приемник. Свойство **DockClientCount** содержит число клиентов. Оба свойства — **DockClients** и **DockClientCount** только для чтения.

Определены соответствующие события, которые позволяют приемнику управлять процессом встраивания. Они генерируются только в том случае, если в компоненте установлено **DockSite = true**.

Первое из этих событий — **OnGetSiteInfo**, возникающее в момент начала перетаскивания и повторяющееся непрерывно в процессе перетаскивания. Заголовок соответствующего обработчика имеет вид:

```
void __fastcall TForm1::Panel1GetSiteInfo(TObject *Sender,
    TControl *DockClient, TRect &InfluenceRect,
    TPoint &MousePos, bool &CanDock)
```

В качестве параметров в обработчик передается **DockClient** — перетаскиваемый объект, **InfluenceRect** — прямоугольник рамки перетаскиваемого объекта, который можно изменять, **MousePos** — положение курсора мыши и **CanDock** — разрешение перетаскивания. Если в обработчике указать **CanDock = false**, это будет означать, что приемник отказывается принимать компонент. Например, вы можете написать в обработчике события **OnGetSiteInfo** панели **Panel1** оператор:

```
CanDock = DockClient != Panel6;
```

Тогда **Panel1** будет принимать любой компонент, кроме **Panel6**.

Еще одно событие — **OnDockOver**, возникающее и повторяющееся, когда компонент-клиент перемещается над компонентом-приемником, точнее, когда в поле приемника войдет курсор мыши, буксирующий клиента. Это событие аналогично событию **OnDragOver** в технологии Drag&Drop. Заголовок соответствующего обработчика имеет вид:

```
void __fastcall TForm1::Panel1DockOver(TObject *Sender,
                                       TDragDockObject *Source, int X, int Y,
                                       TDragState State, bool &Accept)
```

В этом обработчике вы можете, как и в описанном выше, проверить, хотите ли вы, чтобы данный оконный компонент принял перетаскиваемый компонент, определяемый параметром **Source**. Если принимать не надо, то задается значение **Accept = false**.

Например, прием контейнером **Panel1** любой панели, кроме **Panel6**, обеспечивается оператором:

```
Accept = Source->Control != Panel6;
```

Параметр **Accept** будет равен **true** — значению по умолчанию, только если перетаскивается не **Panel6**. Впрочем, того же самого результата вы достигали ранее в обработчике события **OnGetSiteInfo**.

Объект перетаскивания **Source** типа **TDragDockObject** имеет свойство **DockRect** типа **TRect**, описывающее перемещаемую рамку. Вы можете управлять ею. Например, если вы хотите, чтобы ваша панель **Panel5** вела себя наподобие панели Microsoft Office, которая может прижиматься к одной из сторон окна или перемещаться в середине окна в виде квадратной панели, вы можете, установив предварительно в приемнике (в форме вашего приложения или в панели **Panel1**) **DockSite** равным **true**, написать обработчик события **OnDockOver** в виде:

```
int x = ClientOrigin.x;
int y = ClientOrigin.y;
if (Source->Control == Panel5)
{
    if (Source->DockRect.Left <= x)
        Source->DockRect = Rect(x, y, x+25, y+ClientHeight);
    else if (Source->DockRect.Right >= x+ClientWidth)
        Source->DockRect = Rect(x+ClientWidth-25,
                                y, x+ClientWidth, y+ClientHeight);
    else if (Source->DockRect.Top <= y)
        Source->DockRect = Rect(x, y, x+ClientWidth, y+25);
    else if (Source->DockRect.Bottom >= y+ClientHeight)
        Source->DockRect = Rect(x, y+ClientHeight-25,
                                x+ClientWidth, y+ClientHeight);
    else Source->DockRect = Rect(Source->DockRect.Left,
                                Source->DockRect.Top,
                                Source->DockRect.Left+100,
                                Source->DockRect.Top+100);
}
```

В этом коде **x** и **y** — это не параметры **X** и **Y**, передаваемые в обработчик через заголовок процедуры, а координаты левого верхнего угла клиентской области формы. А **ClientWidth** и **ClientHeight** — это ширина и высота ее клиентской области.

Если в процессе буксировки панели **Panel5** ее координаты выходят за пределы клиентской области приемника, то панель вытягивается вдоль соответствующего края приемника, а ее ширина равна 25. При перемещении панели внутри окна она представляется квадратом со стороной 100.

Это будет нормально срабатывать, если для приемника, например, для панели **Panel1**, установить **UseDockManager** равным **false**. В противном случае сначала сработает диспетчер встраивания, растянув рамку клиента, а затем наступит событие **OnDockOver**, так что его обработчик будет иметь дело с уже измененной рамкой, что приведет к ошибке встраивания. Поэтому надо отключить диспетчер встраивания для клиента **Panel5**, не отключая его для остальных клиентов. Это легко сделать, включив в обработчик события **OnGetSiteInfo** оператор

```
Panel1->UseDockManager = DockClient != Panel5;
```

Этот оператор задаст значение **UseDockManager** равным **true** для всех потенциальных клиентов, кроме **Panel5**.

Еще одно событие, возникающее в компоненте-приемнике — **OnDockDrop**. Это событие возникает в момент окончания перетаскивания клиента над приемником. Оно аналогично событию **OnDragDrop** в технологии **Drag&Drop**. Заголовок соответствующего обработчика имеет вид:

```
void __fastcall TForm1::FormDragDrop(TObject *Sender,
                                     TObject *Source, int X, int Y)
```

Этот обработчик позволяет разместить клиента **Source** тем или иным образом в зависимости от его имени, типа, местоположения — координат **X** и **Y**. Например, в этот момент можно что-то изменить в надписях приемника, сигнализируя пользователя о приеме клиента и числе клиентов.

Событие, возникающее в компоненте-приемнике в момент, когда пользователь перетаскивает клиента из приемника — **OnUnDock**. Заголовок соответствующего обработчика имеет вид:

```
void __fastcall TForm1::FormUnDock(TObject *Sender,
                                    TControl *Client, TWinControl *NewTarget, bool &Allow)
```

В обработчик передаются как параметры **Client** — компонент, который был клиентом, **NewTarget** — новый приемник, в который пользователь перетаскивает клиента, и **Allow** — параметр, определяющий, можно ли перетащить клиента. Например, если в вашем тестовом приложении вы поместите в обработчик события **OnUnDock** панели **Panel1** оператор:

```
Allow = Client != Panel2;
```

то, выполняя приложение, обнаружите, что **Panel2**, попав клиентом в **Panel1**, уже не может оттуда выбраться, так как ей это запрещено.

Теперь, когда мы познакомились с основными возможностями технологии **Drag&Doc**, давайте построим что-нибудь более полезное, чем просто игра панелями. Но сначала сохраните ваше тестовое приложение, поскольку оно еще нам потребуется.

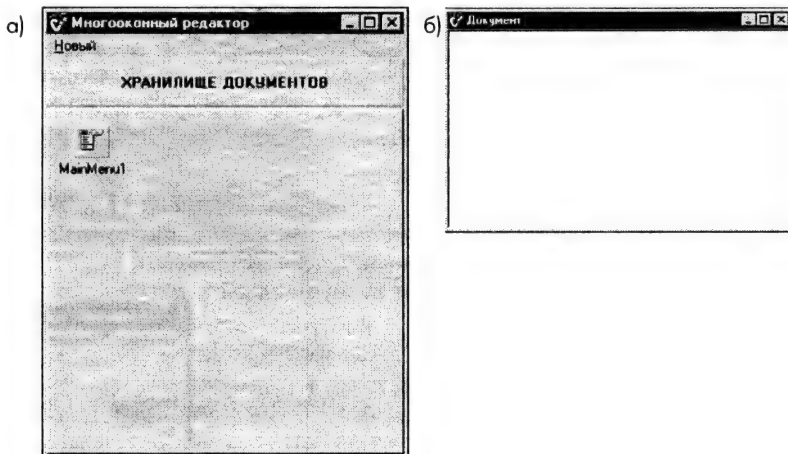
Давайте построим многооконный редактор текстов, имеющий хранилище, в которое можно помещать документы на хранение и затем извлекать из него нужные документы.

Начните новый проект и выполните следующие пункты.

- Назовите форму (свойство **Name**) **Fmain**.
- Разместите на форме компонент **MainMenu** и задайте в нем всего один пункт — Новый, подразумевая под этим создание нового документа.
- Поместите на форму панель, задайте ее свойство **Caption** равным Хранилище документов и задайте **Align = alTop**.
- Поместите на форму компонент **PageControl** и задайте **Align = alClient**, чтобы он занимал вся площадь окна, кроме полосы, занятой панелью Хранилище документов (рис. 4.11 а). Задайте в нем свойство **DockSite** равным **true**. Этот компонент будет служить приемником документов.
- Добавьте в проект еще одну форму, выполнив команду **File | New Form**. Назовите ее **FDoc** (свойство **Name**). Задайте ее свойство **Visible** равным **true**. Установите ее свойство **DragKind** равным **dkDock**, а свойство **DragMode** равным **dmAutomatic**. Эта форма будет служить клиентом компонента **TPageControl** на форме **Fmain**.
- Поместите на форму **FDoc** компонент **Memo**, задав для него **Align = alClient**, чтобы он занял всю форму (рис. 4.11 б). Сотрите в нем текст (свойство **Lines**).
- Выполните команду **Project | Options** и перенесите форму **FDoc** из окна **Auto-create forms** в окно **Available forms**, поскольку она должна создаваться не автоматически.

Рис. 4.11

Формы для приложения, использующего технику Drag&Doc



ски, а при выборе пользователем раздела меню Новый (если эта операция непонятна, см. раздел 4.5.1 рис. 4.14).

- Сохраните проект, назвав модуль с первой формой **Umain**, а со второй — **UDoc**.

Мы создали необходимые нам формы. Осталось написать несколько операторов, чтобы это все работало.

- Выполните для модуля **Umain** команду File | Include Unit Hdr и подключите с ее помощью заголовочный файл **UDoc.h**, чтобы можно было ссылаться на модуль **UDoc**.
- Введите глобальную переменную **LDoc** типа **TList**. Она представляет собой список, в котором будут храниться указатели на создаваемые пользователем формы документов.

```
TList * LDoc;
```

- В обработчик события **OnCreate** формы **Fmain** запишите оператор

```
LDoc = new TList();
```

Этот оператор создает список **LDoc**.

- В обработчик события **OnDestroy** формы **Fmain** запишите оператор

```
delete Ldoc;
```

Этот оператор освобождает память при закрытии приложения.

- Осталось написать обработчик щелчка на разделе меню Новый. Он может иметь следующий вид:

```
TFDoc * New = new TFDoc(this);
LDoc->Add(New);
New->Caption = "Документ "+IntToStr(LDoc->Count);
```

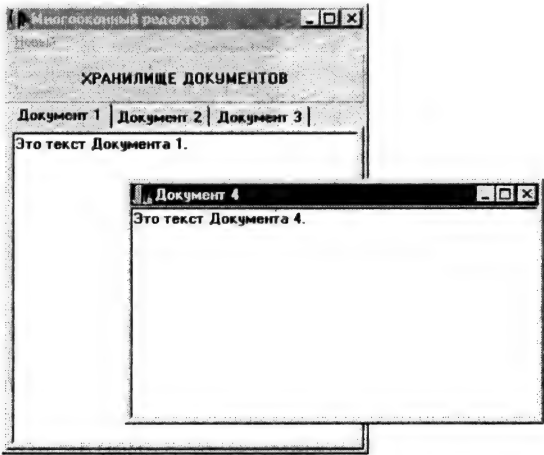
В этом обработчике вводится переменная **New** типа **TFDoc**, соответствующего типу формы **FDoc**. Первый оператор обработчика динамически размещает новую форму **FDoc** и присваивает указатель на нее переменной **New**. Следующий оператор добавляет указатель на эту форму в список **LDoc**. А последний оператор задает заголовок окна формы **FDoc**, равным «Документ ...», где многоточие заменяется на номер, соответствующий числу строк в списке **LDoc**.

Теперь ваше приложение полностью готово. Сохраните его и попробуйте в работе (рис. 4.12). При щелчке на Новый создается новая форма документа, в котором вы можете писать текст. Документы можно помещать в хранилище, в котором

каждому документу автоматически отводится новая страница. Вы можете работать с документом непосредственно в хранилище, а можете изъять его оттуда, превратив опять в отдельное окно.

Рис. 4.12

Приложение, использующее технику Drag&Doc, во время выполнения



Для того, чтобы сделать из вашего приложения законченный продукт, нужна еще, конечно, некоторая техническая работа. Надо бы добавить разделы меню, позволяющие открыть заданный пользователем файл и прочитать его в новый документ, позволяющие сохранить документ в файле, надо бы ввести при закрытии приложения или документа запрос пользователю о необходимости сохранить измененный документ и т.п. Для всех этих операций вам пригодится созданный вами список **LDoc**, который в приведенном упрощенном примере практически не использовался. Но мы не будем этим заниматься, так как все эти манипуляции подробно рассмотрены в разделе 3.8.2 главы 3. А для того, чтобы почувствовать мощь технологии Drag&Drop, достаточно и сделанного вами примера. Причем, заметьте, что для его реализации вам пришлось написать всего несколько операторов.

В заключение рассмотрим еще некоторые свойства и методы, управляющие встраиванием и работой с плавающими окнами.

У оконных компонентов есть свойство **Floating** (только для чтения), которое показывает, является компонент свободно плавающим окном, или размещен в другом оконном компоненте-приемнике.

При переводе ранее размещенного компонента в состояние плавающего окна и при встраивании плавающего окна можно использовать свойства, в которых запоминаются размеры компонента в предыдущем состоянии:

UndockHeight	Высота компонента, которая была в последний раз, когда он отображался плавающим окном
UndockWidth	Ширина компонента, которая была в последний раз, когда он отображался плавающим окном
TBDockHeight	Высота компонента, когда он в последний раз размещался в контейнере вертикально
LRDockWidth	Ширина компонента, когда он в последний раз размещался в контейнере горизонтально

Пользуясь этими свойствами можно восстанавливать при необходимости предшествующие размеры компонента. В частности, по умолчанию после перевода

компонента из размещенного состояния в состояние плавающего окна его размеры восстанавливаются исходя из свойств **UndockHeight** и **UndockWidth**.

Имеется метод **ManualFloat**, который переводит размещенный компонент в состояние плавающего окна. Он объявлен следующим образом:

```
bool __fastcall ManualFloat(const Windows::TRect &ScreenPos);
```

Параметр функции **ScreenPos** определяет положение и размер компонента после его перевода в состояние плавающего окна.

Например, вы можете открыть свой тестовое приложение (рис. 4.10) и ввести в него кнопку, которая будет переводить в состояние плавающего окна, например, панель **Panel3**. Текст обработчика щелчка на такой кнопке может быть следующим:

```
TRect TempRect;
TPoint P;
P = Panel3->ClientToScreen(Point(0, 0));
TempRect.Left = P.x;
TempRect.Top = P.y;
P = Panel3->ClientToScreen(Point(Panel3->UndockWidth,
                               Panel3->UndockHeight));
TempRect.Right = P.x;
TempRect.Bottom = P.y;
Panel3->ManualFloat(TempRect);
```

В этом коде вводится переменная **TempRect**, в которой формируется прямоугольник, определяющий размер и местоположение генерируемого плавающего окна. Этот прямоугольник восстанавливает размер панели **Panel3**, бывший у нее в последний раз, когда она отображалась плавающим окном. Левый верхний угол прямоугольника совпадает с текущим положением панели. Для правильного размещения окна координаты с помощью метода **ClientToScreen** переводятся в координаты экрана. Последний оператор методом **ManualFloat** генерирует плавающее окно.

И последний метод, который мы рассмотрим — **ManualDock**, размещающий компонент в указанном приемнике. Его описание:

```
bool __fastcall ManualDock(TWinControl* NewDockSite,
                          TControl* DropControl, TAlign ControlSide);
```

Параметр **NewDockSite** определяет приемник, в котором должен размещаться клиент. Второй параметр **DropControl** определяет другой компонент в приемнике **NewDockSite**, который должен разместить данного клиента. Как правило, этот параметр задается равным **nil**. Параметр **ControlSide** определяет выравнивание клиента внутри приемника. Если это единственный клиент в приемнике, то значение этого параметра безразлично. Но если там уже есть клиенты, то задание, например, **ControlSide = alLeft** приведет к перестановке, если только данный клиент не самый левый в приемнике.

Можете ввести в свое тестовое приложение кнопку, снабдив ее обработчиком:

```
Panel4->ManualDock(Panel1, NULL, alLeft);
```

Этот обработчик будет размещать **Panel2** в **Panel1**, выравнивая ее по левому краю приемника, если в нем расположены еще какие-нибудь клиенты.

Имеется, однако, особенность, не оговоренная в документации: метод **ManualDock** срабатывает только в случае, если компонент-клиент был ранее в состоянии плавающего окна. Так что вам придется предварительно один раз перевести **Panel2** в это состояние или щелчком на панели, или с помощью кнопки, соответствующей ранее рассмотренному методу **ManualFloat**. Другой способ обойти эту трудность автоматически — ввести соответствующий метод **ManualFloat** в обработчик события формы **OnCreate**.

И еще одно нестандартное использование методов **ManualFloat** и **ManualDock**. Пусть, например, вы хотите иметь форму, содержащую несколько панелей, которые пользователь мог бы перемещать, не переводя их в состояние плавающего окна и не изменяя размеров. Тогда вы можете в форме установить **DockSite = true**, сделать в панелях соответствующие установки свойств **DragKind** в **dkDock** и **DragMode** в **dmAutomatic**, а в событии формы **OnCreate** выполнить для каждой панели методы **ManualFloat** и **ManualDock**, назначив приемником в **ManualDock** форму. Вы получите приложение, в котором пользователь сможет перемещать панели мышью. Это может быть полезно в каких-то приложениях, связанных с проектированием топологии или конструированием. Подобные задачи будут рассмотрены в следующем разделе.

4.4.3 Буксировка компонентов в окне приложения

В ряде случаев в приложениях Windows используется перемещение отдельных компонентов в поле окна или какой-то панели. Это перемещение может быть перемещением в какие-то заранее определенные позиции (это легко осуществляется заданием соответствующих значений свойствам **Left** и **Top**) или осуществляться непрерывной буксировкой компонента с помощью мыши. Простой пример этого — буксировка компонентов по площади формы в среде разработки C++Builder. Подобные задачи возникают достаточно часто в технических приложениях, связанных с компоновкой какого-то устройства, с формированием каких-то схем (например, электрических) и т.п.

Опробовать различные способы буксировки можно в тестовом приложении, подобном приведенному на рис. 4.13. Этот пример имеется на прилагаемом к книге диске. В нем на форме размещено четыре компонента **Image**, в которые загружены какие-то изображения или части изображения (о загрузке в **Image** изображений и о работе с этими компонентами см. раздел 5.1.1). Если хотите воспроизвести пример, подобный рис. 4.13, в котором на отдельных компонентах, как на детских кубиках, воспроизведены фрагменты единого изображения, то можете в обработчик события формы **OnCreate** вставить следующий код:

```
TImage * Pict = new TImage(Form1);
Pict->AutoSize = true;
/* В следующем операторе вместо ...
   надо указать имя файла */
Pict->Picture->LoadFromFile("...");
Image1->Canvas->CopyRect(Image1->ClientRect, Pict->Canvas,
    Rect(0,0,Pict->Width / 2,Pict->Height / 2));
Image2->Canvas->CopyRect(Image2->ClientRect, Pict->Canvas,
    Rect(Pict->Width / 2,0,Pict->Width,Pict->Height / 2));
Image3->Canvas->CopyRect(Image3->ClientRect, Pict->Canvas,
    Rect(0,Pict->Height / 2,Pict->Width / 2,Pict->Height));
Image4->Canvas->CopyRect(Image4->ClientRect, Pict->Canvas,
    Rect(Pict->Width / 2,Pict->Height / 2,
    Pict->Width,Pict->Height));
delete Pict;
```

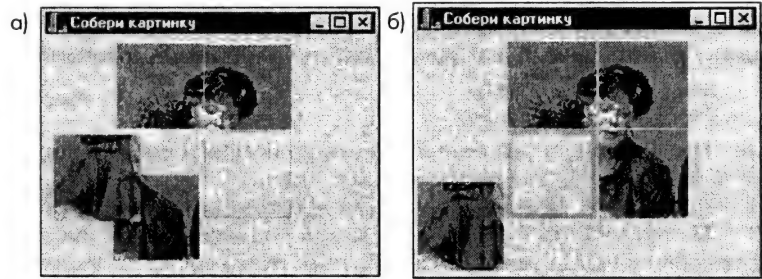
Сейчас мы не будем анализировать этот код. Он станет вам понятен после изучения главы 5. В компонентах **Image** свойство **AutoSize** должно быть установлено в **true**.

Начнем рассмотрение на этом примере приемов буксировки. Все приведенные далее обработчики событий записаны в общем виде. Так что вы можете применять их одновременно ко всем вашим компонентам **Image**, а можете к разным компонентам применить разные методы, чтобы легче их было сравнивать друг с другом.

Все методы используют несколько глобальных переменных, объявление которых надо поместить в модуле вне каких-либо процедур:

Рис. 4.13

Приложение,
демонстрирующее
буксировку компонентов



```
int X0, Y0;
bool move = false;
```

Переменная **move** определяет режим буксировки. Она будет устанавливаться в **true** в начале буксировки и сбрасываться в **false** в конце. Поэтому по значению **move** можно будет различать перемещения мыши с буксировкой и без нее. Переменные **X0** и **Y0** потребуются нам для запоминания координат курсора мыши.

Один из возможных вариантов решения нашей задачи — буксировка самого компонента. Буксировка начинается при нажатии левой кнопки мыши на соответствующем компоненте **Image**. Поэтому начало определяется событием **OnMouseDown**, обработчик которого имеет вид:

```
void __fastcall TForm1::Image1MouseDown(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
    if(Button != mbLeft) return;
    X0 = X;
    Y0 = Y;
    move = true;
    ((TControl *)Sender)->BringToFront();
}
```

Сначала в этой процедуре проверяется, нажата ли именно левая кнопка мыши (равен ли параметр **Button** значению **mbLeft**, обозначающему левую кнопку). Затем в переменных **X0** и **Y0** запоминаются координаты мыши **X** и **Y** в этот момент времени. Задается режим буксировки — переменная **move** устанавливается в **true**. Последний оператор выдвигает методом **BringToFront** компонент, в котором произошло событие — **((TControl *)Sender)**, на передний план. Это позволит ему в дальнейшем перемещаться поверх других аналогичных компонентов. Данный оператор не обязателен, но если его не записать, то в процессе буксировки перемещаемый компонент может оказаться под другими компонентами.

Во время буксировки компонента работает его обработчик события **OnMouseMove**, имеющий вид:

```
void __fastcall TForm1::Image1MouseMove(TObject *Sender,
    TShiftState Shift, int X, int Y)
{
    if (move)
    {
        TImage * Im = (TImage *)Sender;
        Im->SetBounds(Im->Left + X - X0,
            Im->Top + Y - Y0, Im->Width, Im->Height);
    }
}
```

Он изменяет с помощью метода **SetBounds** координаты левого верхнего угла на величину сдвига курсора мыши (**X — X0** для координаты **X** и **Y — Y0** для координаты **Y**). Тем самым поддерживается постоянное расположение точки курсора в системе координат компонента, т.е. компонент смещается вслед за курсором. Ши-

рина **Width** и высота **Height** компонента остаются неизменными. Изменение координат можно было бы задавать непосредственным изменением свойств **Left** и **Top** :

```
Im->Left += X - X0;
Im->Top += Y - Y0;
```

Но поскольку эти операторы выполняются поочередно, то компонент будет делать каждый раз два перемещения: сначала по горизонтали, а потом по вертикали. Это приведет к заметному на глаз дрожанию изображения.

По окончании буксировки, когда пользователь отпустит кнопку мыши, наступит событие **OnMouseUp**. Обработчик этого события должен содержать всего один оператор:

```
move = false;
```

указывающий приложению на окончание буксировки. Тогда при последующих событиях **OnMouseMove** их обработчик, приведенный ранее, перестанет изменять координаты компонента.

Рассмотренный вариант буксировки не свободен от недостатков. Основной из них — некоторое дрожание изображения при перемещении, что выглядит не очень приятно. Устранить этот недостаток позволяет другой вариант, заключающийся в буксировке не самого компонента, а его контура (этот вариант вы можете видеть на рис. 4.13 а). При этом отмеченных выше неприятных явлений не наблюдается, поскольку сам компонент перемещается только один раз — в момент окончания буксировки, когда требуемое положение уже выбрано. В этом варианте используются методы рисования на канве, которые подробно рассматриваются в главе 5 в разделе 5.1.3. Для их применения нам потребуется еще одна глобальная переменная:

```
TRect rec;
```

Это объявление следует добавить к приведенным ранее объявлениям **move**, **X0** и **Y0**. Переменная **rec** будет использоваться для запоминания положения перемещаемого контура компонента.

Начинается процесс буксировки, как и ранее, с события **OnMouseDown**, которое обрабатывается процедурой вида:

```
void __fastcall TForm1::Image2MouseDown(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
    if(Button != mbLeft) return;
    X0 = X;
    Y0 = Y;
    rec = ((TControl *)Sender)->BoundsRect;
    move = true;
}
```

Эта процедура отличается от рассмотренной ранее только оператором

```
rec = ((TControl *)Sender)->BoundsRect;
```

который запоминает в переменной **rec** исходное положение компонента. В процедуре отсутствует также оператор **BringToFront**, поскольку сам компонент не будет перемещаться и не приходится заботиться о том, чтобы он оказался поверх аналогичных компонентов.

При дальнейшем перемещении курсора мыши срабатывает обработчик события **OnMouseMove**, имеющий вид:

```
void __fastcall TForm1::Image2MouseMove(TObject *Sender,
    TShiftState Shift, int X, int Y)
{
    if(! move) return;
    Canvas->DrawFocusRect(rec);
    rec.left += X - X0;
    rec.right += X - X0;
```

```

rec.top += Y - Y0;
rec.bottom += Y - Y0;
X0 = X;
Y0 = Y;
Canvas->DrawFocusRect(rec);
}

```

В этой процедуре перерисовывается и сдвигается только прямоугольник контура компонента с помощью метода **DrawFocusRect**. Первое обращение к этому методу стирает прежнее изображение контура, поскольку повторная прорисовка того же изображения по операции **ИЛИ (or)** стирает нанесенное ранее изображение (см. главу 5, раздел 5.1.5). Затем изменяются значения, хранимые в переменной **rec**, и той же функцией **DrawFocusRect** осуществляется прорисовка сдвинутого прямоугольника. При этом сам компонент остается на месте.

Когда пользователь отпускает кнопку мыши, наступает событие **OnMouseUp** и срабатывает следующая процедура:

```

void __fastcall TForm1::Image4MouseUp(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
    Canvas->DrawFocusRect(rec);
    ((TControl *)Sender)->SetBounds(
        rec.Left + X - X0, rec.Top + Y - Y0,
        ((TControl *)Sender)->Width,
        ((TControl *)Sender)->Height);
    ((TControl *)Sender)->BringToFront();
    move = false;
}

```

Первый ее оператор стирает последнее изображение контура, а второй оператор перемещает компонент в новую позицию. Оператор **BringToFront** не является обязательным. Он проявит себя только в случае, если перемещенный компонент ляжет поверх другого аналогичного компонента.

Приведенный алгоритм исключает мерцание изображения, так как само перемещение компонента осуществляется только в момент отпускания кнопки мыши. К тому же при этом в обработчике события **OnMouseUp** легко предусмотреть какие-то условия отказа от перемещения (например, нажатую клавишу **Alt**). Для этого достаточно в приведенный выше обработчик добавить проверку нажатых клавиш:

```

Canvas->DrawFocusRect(rec);
if(! Shift.Contains(ssAlt))
{
    ((TControl *)Sender)->SetBounds(
        rec.Left + X - X0, rec.Top + Y - Y0,
        ((TControl *)Sender)->Width,
        ((TControl *)Sender)->Height);
    ((TControl *)Sender)->BringToFront();
}
move = false;

```

В этом случае, если пользователь перед завершением буксировки нажмет клавишу **Alt**, компонент не переместится. Это полезно в некоторых приложениях, связанных с каким-то проектированием топологии или размещения предметов: прикинув новое положение компонента пользователь может тут же вернуться к предыдущему, если новое расположение неудачно. С другой стороны, для некоторых задач буксировка только контура мало информативна. Например, при проектировании какой-то мозаики перемещение контура не даст представления о том, подойдет ли компонент по своему виду к данному месту. В этих случаях предпочтительнее буксировка всего изображения.

Описанный выше способ перемещения изображения контура компонента можно осуществить иначе с использованием техники **Drag&Dock** и методов **ManualFloat** и **ManualDock**, описанных в разделе 4.4.2. Чтобы опробовать этот подход, установите в форме свойство **DockSite** равным **true** и сделайте в перемещаемых компонентах (в нашем примере — в **Image**) установку свойств **DragKind** в **dkDock** и **DragMode** в **dmAutomatic**. Это надо сделать по крайней мере в двух компонентах. Если перемещаемым будет только один компонент, описываемый метод не сработает. В начале работы приложения надо выполнить для каждого компонента, который в дальнейшем может перемещаться, методы **ManualFloat** и **ManualDock**, назначив приемником в **ManualDock** форму. Для этого можно вставить в обработчик события формы **OnCreate**, например, следующие операторы:

```
Image3->ManualFloat (Rect (Form1->Left+Image3->Left,  
                           Form1->Top+Image3->Top,  
                           Form1->Left+Image3->Left+Image3->Width,  
                           Form1->Top+Image3->Top+Image3->Height));  
Image3->ManualDock (Form1, NULL, alLeft);
```

Аналогичные операторы с заменой имени компонента **Image3** надо вставить и для других перемещаемых компонентов.

И это все. Вам не надо писать никаких обработчиков событий компонентов. Вы получили приложение, в котором пользователь может перемещать мышью ваши изображения. При этом в процессе буксировки перемещается только контур изображения в виде утолщенной рамки (рис. 4.13 б). Сам компонент перемещается на новое место только в конце буксировки. Если хотите, то можете в обработчик события **OnEndDock** внести оператор

```
((TControl *)Sender)->BringToFront();
```

который обеспечит в момент переноса размещение компонента поверх других.

4.5 Формы

4.5.1 Управление формами

Основным элементом любого приложения является форма — контейнер, в котором размещаются другие визуальные и не визуальные компоненты. Рассмотрим некоторые свойства, методы и события, присущие формам.

Обычно сколько-нибудь сложное приложение содержит несколько форм. Включение в проект новой формы осуществляется командой **File | New Form** или другими способами, подробно описанными в разделе 2.5.1. По умолчанию все формы создаются автоматически при запуске приложения и первая из введенных в приложение форм считается главной. Главная форма отличается от прочих рядом свойств. Во-первых, именно этой форме передается управление в начале выполнения приложения. Во-вторых, закрытие пользователем главной формы означает завершение выполнения приложения. В-третьих, главная форма так же, как и любая другая, может быть спроектирована невидимой, но если все остальные формы зарыты, то главная форма становится в любом случае видимой (иначе пользователь не смог бы продолжать работать с приложением и даже не смог бы его завершить).

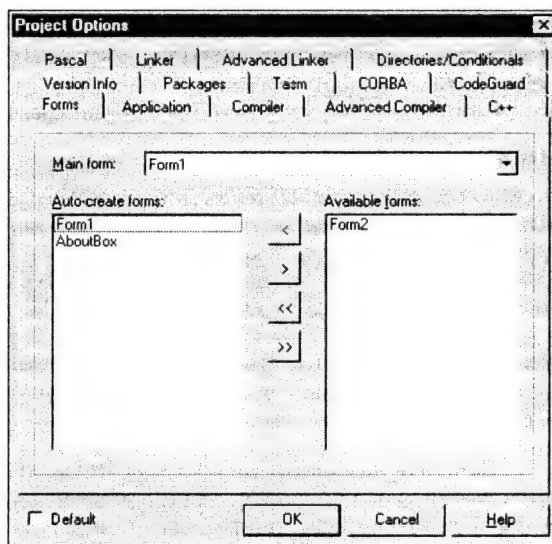
Указанные выше условия, принятые по умолчанию (первая форма — главная, все формы создаются автоматически), могут быть изменены. Главной в вашем приложении может быть вовсе не та форма, которая была спроектирована первой. Не стоит также в общем случае все формы делать создаваемыми автоматически. В приложении могут быть предусмотрены формы (например, формы для установки различных опций), которые требуются далеко не в каждом сеансе работы с приложением. Было бы варварским расточительством создавать на всякий случай такие

формы автоматически при каждом запуске приложения и занимать под них память. А в приложениях MDI дочерние формы в принципе не могут быть автоматически создаваемыми, так как число таких форм определяет пользователь во время работы приложения, создавая каждую новую форму командой типа Новое окно.

Изменить принятые по умолчанию условия относительно форм можно в окне Опций проекта, которое вызывается, например, командой Project | Options главного меню. В открывшемся окне Опций проекта (Project Options) надо выбрать страницу Forms, представленную на рис. 4.14.

Рис. 4.14

Страница Forms окна Опций проекта



В верхнем выпадающем списке Main forms можно выбрать главную форму среди имеющихся в проекте. Пользуясь двумя нижними окнами можно установить, какие формы должны создаваться автоматически, а какие не должны. Например, если надо исключить форму **Form2** из списка автоматически создаваемых, то надо выделить ее в левом окне (Auto-create forms) и с помощью кнопки со стрелкой, направленной вправо, переместить в правое окно доступных форм (Available forms).

Для каждой автоматически создаваемой формы C++Builder добавляет в файл программы соответствующий оператор ее создания методом **CreateForm**. Это можно увидеть, если выполнить команду View | Project Source и просмотреть появившийся в окне Редактора Кода файл проекта. Для примера, показанного на рис. 4.14, он будет содержать следующие выполняемые операторы:

```
Application->Initialize();  
Application->CreateForm(__classid(TForm1), &Form1);  
Application->CreateForm(__classid(TAboutBox), &AboutBox);  
Application->Run();
```

Первый из них инициализирует приложение, второй и третий создают соответствующие формы, а последний начинает выполнение приложения.

Для форм, которые были исключены из списка автоматически создаваемых, аналогичный метод **CreateForm** надо выполнить в тот момент, когда форма должна быть создана. Например, чтобы создать в нужный момент при выполнении кода в модуле формы **Form1** форму **Form2**, надо выполнить оператор:

```
Application->CreateForm(__classid(TForm2), &Form2);
```

Форма **Form2** будет создана и, если ее свойство **Visible** установлено в **true**, то пользователь в тот же момент увидит ее на экране. Только для того, чтобы это сра-

ботало, надо в модуль формы **Form1** включить заголовочный файл модуля формы **Form2**. Если имя этого файла **Unit2.h**, то можно вручную включить в модуль формы **Form1** директиву препроцессора

```
#include "Unit2.h"
```

То же самое можно сделать, выполнив команду главного меню File | Include Unit Hdr и выбрав в появившемся диалоговом окне имя нужного файла. Тогда указанная выше директива препроцессора будет записана в модуль автоматически.

В момент создания формы возникает последовательность событий:

Событие	Источник события	Описание
OnCreate	форма	создание формы и всех управляемых ею компонентов
OnShow	форма	после этого события форма становится видимой
OnActivate	форма	управление (фокус) передается данной форме
OnEnter	первый компонент в последовательности табуляции формы	фокус передается компоненту, первому в последовательности табуляции
OnResize	форма	переустанавливаются размеры формы
OnPaint	форма	прорисовка изображения формы

Обратите внимание на событие **OnCreate**. Обработка этого события широко используется для настройки каких-то компонентов формы, создания списков и т.д. Это событие для каждой формы происходит только один раз. Все остальные события могут в процессе выполнения приложения неоднократно повторяться.

В нужный момент форму можно сделать видимой методами **Show** или **ShowModal**. Последний метод открывает форму как модальную. Это означает, что управление передается этой форме и пользователь не может передать фокус другой форме данного приложения до тех пор, пока он не закроет модальную форму. Более подробное описание модальных форм и примеры работы с ними будут рассмотрены позднее в разделе 4.5.2.

Методы **Show** и **ShowModal** можно применять только к невидимой в данный момент формы. Если нет уверенности, что форма в данный момент видима, то прежде, чем применять эти методы, следует проверить свойство **Visible** формы. При выполнении методов **Show** или **ShowModal** возникает событие формы **onShow**. Это событие возникает до того момента, как форма действительно станет видимой. Поэтому обработку события **onShow** можно использовать для настройки каких-то компонентов открываемой формы. Отличие от упомянутой ранее настройки компонентов в момент события **onCreate** заключается в том, что событие **onCreate** наступает для каждой формы только один раз в момент ее создания, а события **onShow** наступают каждый раз, когда форма делается видимой. Так что при этом в настройке можно использовать какую-то оперативную информацию, возникающую в процессе выполнения приложения.

Методом **Hide** форму в любой момент можно сделать невидимой. В этот момент в ней возникает событие **onHide**.

Необходимо напомнить, что для выполнения методов **CreateForm**, **Show**, **ShowModal**, **Hide** и вообще для обмена любой информацией между формами модули соответствующих форм должны видеть друг друга, а для этого в них надо включать описанные выше директивы препроцессора **#include**. Например, если форма в модуле **Unit1** должна управлять формой в модуле **Unit2**, то в модуль **Unit1** должна

быть включена инструкция `#include "Unit2.h"`. А если к тому же форма в модуле **Unit2** должна пользоваться какой-то информацией, содержащейся в модуле **Unit1**, то в модуль **Unit2** должна быть включена инструкция `#include "Unit1.h"`.

Закрыть форму можно методом **Close**. При этом в закрывающейся форме возникает последовательность событий, которые можно обрабатывать. Их назначение — проверить возможность закрытия формы и указать, что именно подразумевается под закрытием формы. Проверка возможности закрытия формы необходима, например, для того, чтобы проанализировать, сохранил ли пользователь документ, с которым он работал в данной форме и который изменял. Если не сохранил, приложение должно спросить его о необходимости сохранения и, в зависимости от ответа пользователя, сохранить документ, закрыть приложение без сохранения или вообще отменить закрытие.

Рассмотрим последовательность событий, возникающих при выполнении метода **Close**.

Первым возникает событие **onCloseQuery**. В его обработчик передается по ссылке булева переменная **CanClose**, определяющая, должно ли продолжаться закрытие формы. По умолчанию **CanClose** равно **true**, что означает продолжение закрытия. Но если из анализа текущего состояния приложения или из ответа пользователя на запрос о закрытии формы следует, что закрывать ее не надо, параметру **CanClose** должно быть присвоено значение **false**. Тогда последующих событий, связанных с закрытием формы не будет.

Например, пусть в приложении имеется окно редактирования **RichEdit1**, в котором свойство **Modified** указывает на то, был ли изменен пользователем текст в этом окне с момента его последнего сохранения. Тогда обработчик события **onCloseQuery** может иметь вид:

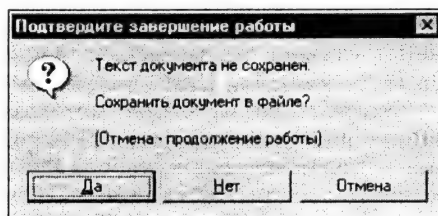
```
void __fastcall TForm1::FormCloseQuery(TObject *Sender,
                                       bool &CanClose)
{
    if (RichEdit1->Modified)
    {
        int res = Application->MessageBox(
            "Текст документа не сохранен. \n\n"
            "Сохранить документ в файле?\n\n"
            "(Отмена — продолжение работы)",
            "Подтвердите завершение работы",
            MB_YESNOCANCEL + MB_ICONQUESTION);

        switch (res)
        {
            case IDYES:    MSaveClick(Sender);
                          break;
            case IDCANCEL: CanClose = false;
        }
    }
}
```

В приведенном обработчике вызывается методом **Application->MessageBox** (см. его подробное описание в главе 15 в разделе 15.7.2.3) диалоговое окно, показанное на рис. 4.15. Если пользователь ответит «Да», то будет выполнена описанная в приложении процедура сохранения **MSaveClick**. Если пользователь ответит «Нет», то никаких действий в структуре **switch** производиться не будет. В обоих случаях после выхода из обработчика события значение **CanClose** останется равным своему значению по умолчанию **true** и процесс закрывания формы будет продолжен. Но если пользователь при запросе ответит «Отмена» или нажмет клавишу Esc, то значение **CanClose** станет равно **false** и окно не закроется. Этот обработчик сработает при любой попытке пользователя закрыть приложение: нажатии в нем кнопки или раздела меню Выход, нажатии кнопки системного меню в полосе заголовка окна и т.п.

Рис. 4.15

Диалоговое окно с запросом о завершении работы



Если обработчик события **onCloseQuery** отсутствует или если в его обработчике сохранено значение **true** параметра **CanClose**, то следом наступает событие **OnClose**. В обработчик этого события передается по ссылке переменная **Action**, которой можно задавать значения:

caNone	Не закрывать форму. Это позволяет и в обработчике данного события еще отказаться от закрытия формы
caHide	При этом значении (оно принято по умолчанию для форм, не являющихся главными и дочерними в приложениях MDI) закрыть форму будет означать сделать ее невидимой. Для пользователя она исчезнет с экрана, однако вся хранящаяся в форме информация сохранится. Она может использоваться другими формами или той же самой формой, если она снова будет сделана видимой
caMinimize	При этом значении закрыть форму будет означать свернуть ее до пиктограммы. Как и в предыдущем случае, вся информация в форме будет сохранена
caFree	При этом значении закрыть форму будет означать уничтожение формы и освобождение занимаемой ею памяти. Вся информация, содержащаяся в форме, будет уничтожена. Если эта форма в дальнейшем потребуется еще раз, ее надо будет создавать методом CreateForm

Не все значения **Action** допустимы для любых форм. Например, для дочерних форм в приложении MDI возможны значения только **caNone** и **caMinimize**.

Если в обработчике события **OnClose** задано значение **Action**, равное **caFree**, то при освобождении памяти возникает еще одно последнее событие — **OnDestroy**. Оно обычно используется для очистки памяти от тех объектов, которые автоматически не уничтожаются при закрытии приложения.

Начиная с C++Builder 4 формы имеют свойство **OldCreateOrder**, определяющее моменты событий **OnCreate** и **OnDestroy**. Если это свойство установлено в **false** (значение по умолчанию), то событие **OnCreate** наступает после того, как закончили работу все конструкторы компонентов, содержащихся на форме, а событие **OnDestroy** наступает прежде, чем вызывается какой-либо деструктор. При **OldCreateOrder = true**, что соответствует поведению компонентов в C++Builder 3 и более ранних, событие **OnCreate** наступает при выполнении конструктора **TCustomForm**, а событие **OnDestroy** наступает при выполнении деструктора **TCustomForm**.

4.5.2 Модальные формы

Открытие форм как модальных используется в большинстве диалоговых окон. Модальная форма приостанавливает выполнение вызвавшей ее процедуры до тех пор, пока пользователь не закроет эту форму. Модальная форма не позволяет также пользователю переключить фокус курсором мыши на другие формы данного приложения, пока форма не будет закрыта. Так что пользователь должен выполнить предложенные ему действия прежде, чем продолжить работу.

Модальной может быть сделана любая форма, если она делается видимой методом **ShowModal**. Если та же самая форма делается видимой методом **Show**, то она не будет модальной.

Поведение модальной формы определяется ее основным свойством **ModalResult**. Это свойство доступно только во время выполнения приложения. При открытии формы методом **ShowModal** сначала свойство **ModalResult** равно нулю. Как только при обработке каких-то событий на форме свойству **ModalResult** будет присвоено положительное значение от 1 до 8, модальная форма закроется. А значение ее свойства **ModalResult** можно будет прочитать как результат, возвращаемый методом **ShowModal**. Таким образом программа, вызвавшая модальную форму, может узнать, что сделал пользователь, работая с этой формой, например, на какой кнопке он щелкнул.

В C++Builder предопределены некоторые константы, облегчающие трактовку результатов, полученных при закрытии модальной формы:

Численное значение ModalResult	Константа	Пояснение
0	mrNone	
1	mrOk или idOK	Закрытие модальной формы нажатием кнопки OK
2	mrCancel или idCancel	Закрытие модальной формы нажатием кнопки Cancel, или методом Close , или нажатием кнопки системного меню в полосе заголовка окна
3	mrAbort или idAbort	Закрытие модальной формы нажатием кнопки Abort
4	mrRetry или idRetry	Закрытие модальной формы нажатием кнопки Retry
5	mrIgnore или idIgnore	Закрытие модальной формы нажатием кнопки Ignore
6	mrYes или idYes	Закрытие модальной формы нажатием кнопки Yes
7	mrNo или idNo	Закрытие модальной формы нажатием кнопки No
8	mrAll	Закрытие модальной формы нажатием кнопки All

Все приведенные выше пояснения значений **ModalResult** (кроме значений 0 и 2) носят чисто условный характер. В своем приложении вы вольны трактовать ту или иную величину **ModalResult** и соответствующие константы как вам угодно.

Требуемые значения **ModalResult** можно задавать в обработчиках соответствующих событий в компонентах модальной формы. Однако при использовании кнопок можно обойтись и без подобных обработчиков. Дело в том, что кнопки типа **TButton** и **TBitBtn** имеют свойство **ModalResult**, по умолчанию равное **mrNone**. Для кнопок, расположенных на модальной форме, значение этого свойства можно изменить и тогда не потребуется вводить каких-либо обработчиков событий при щелчке на них. В кнопках **BitBtn** при свойстве **Kind**, не равном **bkCustom**, заложены по умолчанию значения **ModalResult**, соответствующие назначению той или иной кнопки.

Ниже приведен пример использования модальных форм.

4.5.3 Пример приложения с модальными формами заставки и запроса пароля

Во многих больших приложениях при их запуске сначала на экране появляется форма-заставка, содержащая логотип приложения и сведения о программе и ее разработчике. Назначение этой формы чаще всего заключается в том, чтобы обеспечить начальную загрузку и настройку программы. Тогда эта форма должна закрываться не раньше, чем закончатся эти операции. Но иногда эта форма носит чисто информационный характер. В этих случаях желательно, чтобы она немедленно закрывалась при любых действиях пользователя и даже закрывалась через какое-то время без каких-либо действий со стороны пользователя. Именно такую форму-заставку мы и попробуем создать.

Помимо формы-заставки нередко в приложениях, особенно в тех, которые работают с базами данных, в начале работы приложения появляется форма с запросом пароля. При неверном пароле приложение закрывается, не позволяя пользователю работать с ним.

Формы-заставки и формы запроса пароля могут быть реализованы множеством различных способов. Рассмотрим один из них.

1. Откройте в C++Builder новое приложение (File | New Application). Пусть открывшаяся форма будет главной в нашем приложении (вместо такой пустой формы вы можете взять любое разработанное вами ранее приложение и добавлять форму-заставку и форму запроса пароля в него). Назовите для определенности главную форму приложения **FMain**.
2. Добавьте в приложение новую форму (File | New Form). Пусть это будет ваша форма-заставка. Назовите ее **FLog**. Ее свойство **BorderStyle** надо сделать равным **bsNone** (см. раздел 4.1.3), чтобы в окне этой формы отсутствовала полоса заголовка. Вы можете поместить на форме какой-то рисунок (разместить компонент **Image** и вставить в его свойство **Picture** желаемый рисунок), надписи и т.п. В простейшем случае поместите в центре формы метку **Label** и напишите в ней какой-то текст. Размер формы-заставки задайте небольшим, меньшим, чем обычные окна приложения. Свойство **Position** следует сделать равным **poScreenCenter**, чтобы форма появлялась в центре экрана.
3. Теперь напишите обработчики событий, которые при любом действии пользователя закрывали бы форму. Щелкните на форме, чтобы в Инспекторе Объектов открылись относящиеся к ней страницы (если у вас форма накрыта панелями или рисунками, то, щелкнув на них, нажимайте клавишу Esc до тех пор, пока в Инспекторе Объектов не откроются страницы, относящиеся к форме). Перейдите в Инспекторе Объектов на страницу событий, выберите событие **onKeyDown** и напишите для него обработчик, состоящий из одного оператора —

Close(). Аналогичный обработчик напишите для события **onMouseDown**. Если на форме у вас имеются метки, компоненты **Image** и др., то выделите их все, задайте в событии **onMouseDown** ссылку на тот же обработчик, что вы сделали для формы, а в форме поставьте свойство **KeyPreview** в **true**, чтобы форма перехватывала все связанные с нажатием клавиш события компонентов.

Теперь форма будет закрываться при нажатии пользователем любой клавиши или кнопки мыши. Давайте сделаем так, чтобы и при отсутствии каких-то действий со стороны пользователя форма закрывалась сама, например, через 5 секунд.

4. Добавьте на форму компонент **Timer** со страницы **System**. Это невидимый компонент, который может отсчитывать интервалы времени (см. раздел 3.5.6). Интервал задается в свойстве компонента **Interval** в миллисекундах. Задайте его равным 5000. Единственное событие таймера — **onTimer**, наступающее по истечении заданного интервала времени. Напишите в обработчике этого события все тот же единственный оператор **Close()**.

Теперь при любом действии и даже бездействии пользователя форма-заставка будет закрываться. Но что это означает? По умолчанию закрыть форму значит сделать ее невидимой. Однако, форма-заставка не нужна после того, как она будет предъявлена пользователю в первый момент выполнения приложения. Хранить все время в памяти эту уже ненужную форму не имеет смысла. Поэтому в форме надо предусмотреть, чтобы закрытие формы означало ее удаление из памяти и освобождение памяти для чего-нибудь более полезного. Для этого надо сделать следующее.

5. В событие формы **OnClose** вставьте оператор:

```
Action = caFree;
```

Как указывалось в предыдущих разделах, этот оператор приводит к уничтожению объекта формы и освобождению занимаемой формой памяти.

Форма-заставка готова к использованию. Проверьте только, имеет ли ее свойство **Visible** значение **false**. Это важно, поскольку только невидимую форму можно открыть методом **ShowModal**. В главной форме свойство **Visible** тоже должно иметь значение **false**.

Теперь можно записать в главной форме оператор **ShowModal**. Но чтобы это можно было сделать, необходимо сослаться в модуле главной формы на модуль формы-заставки.

6. Сохраните проект, дав файлу модуля главной формы имя **UMain**, а файлу модуля формы-заставки имя **UFLog**. Добавьте в модуль **UMain** директиву компилятора **#include "UFLog.h"** или выполните для модуля **UMain** команду **File | Include Unit Hdr** и укажите в диалоге имя включаемого модуля **UFLog**.

Сохранение необходимо сделать, чтобы модули обрели свои окончательные имена. Иначе если вы сделаете ссылку, пока модули имеют имена по умолчанию (**Unit1** и **Unit2**), а потом при сохранении дадите им более осмысленные имена (это всегда желательно делать), то прежние ссылки на модули окажутся неверными и вам придется их переделывать.

Теперь осталось написать в модуле **UMain** обработчик события формы **OnShow**.

7. Напишите в модуле **UMain** обработчик события формы **OnShow**, состоящий из одного оператора:

```
FLog->ShowModal();
```

Событие **OnShow** наступает перед тем, как форма становится видимой. Поэтому в момент выполнения указанного оператора она еще не видна. Оператор открывает форму **FLog** как модальную, передает ей управление и дальнейшее выполне-

ние программы в модуле **UMain** останавливается до тех пор, пока модальная форма не будет закрыта. После закрытия модальной формы выполнение программы продолжится и главная форма станет видимой.

Можете сохранить проект, запустить приложение и убедиться, что все работает правильно.

Теперь добавим в приложение форму запроса пароля. Реальная форма такого типа должна предлагать пользователю ввести свое имя и пароль, сравнивать введенные значения с образцами, хранящимися где-то в системе, при неправильном пароле давать возможность пользователю поправиться. Если пользователь так и не может ввести правильный пароль, форма должна закрыть приложение, не допуская к нему пользователя. При правильном пароле после закрытия формы запроса должна открыться главная форма приложения. Все это не трудно сделать, но мы упростим задачу, чтобы не отвлекаться от главного — взаимодействия форм в приложении. Будем использовать всего один пароль, который непосредственно укажем в соответствующем операторе программы. И не будем давать пользователю возможности исправить введенный пароль.

8. Добавьте к приложению новую форму. Назовите ее **FPSW** и сохраните ее модуль в файле с именем **UPSW**. Уменьшите размер формы до разумных пределов, поскольку она будет содержать всего одно окно редактирования. Установите свойство формы **BorderStyle** равным **bsDialog**, свойство **Position** равным **poScreenCenter**. В свойстве **Caption** напишите «Введите пароль и нажмите Enter». Эта надпись будет служить приглашением пользователю.
9. Поместите в центре формы окно редактирования **Edit**, в котором пользователь будет вводить пароль. Очистите его свойство **Text**. Задайте в свойстве **PasswordChar** символ "*". В обработчике события **OnKeyDown** этого компонента запишите оператор:

```
if (Key == VK_RETURN) Close();
```

Этот оператор анализирует нажатую клавишу (подробнее об этом смотрите в разделе 4.3.2.2). Если нажата клавиша Enter, то считается, что пользователь завершил ввод пароля, и форма закрывается. Анализ введенного пароля откладывается до события **OnClose**. Дело в том, что, несмотря на приглашение в заголовке окна нажать после ввода пароля Enter, пользователь может проигнорировать это и закрыть окно, например, системной кнопкой. Если он до этого ввел правильный пароль, то несправедливо не проверить его, наказывая таким образом пользователя за непослушание. Вообще, если вы хотите создать для пользователя дружелюбный интерфейс, поменьше диктуйте ему последовательность действий. Лучше исходите из правила: «Пользователь всегда прав». И старайтесь предусмотреть реакцию на его самые неразумные (с вашей точки зрения) действия.

10. В обработчике события **OnClose** формы **FPSW** напишите оператор

```
if (Edit1->Text == "1") ModalResult = 6;
```

Этот оператор сличает введенный текст с паролем. В данном операторе для упрощения непосредственно указан правильный пароль — символ '1'. Если введен правильный пароль, то свойству **ModalResult** присваивается некоторое условное число — 6 (можно было бы выбрать и любое другое допустимое число, кроме 0 и 2). Если пароль неправильный, то оставляется значение **ModalResult** = 2 (**mrCancel**), которое автоматически присваивается при любой попытке закрыть форму. В обоих случаях форма закрывается, так как задание отличного от нуля положительного значения **ModalResult** равносильно закрытию формы. Но при правильном пароле значение **ModalResult** будет равно 6, а при неправильном — 2.

На этом проектирование формы запроса пароля закончено. Теперь запишем в модуле главной формы **UMain** оператор, показывающий пользователю эту форму и анализирующий ответ пользователя. Для этого надо выполнить следующие операции.

11. В модуле **UMain**, надо, как и ранее, добавить ссылку на модуль **UPSW**, а в обработчике события **OnShow** после ранее введенного оператора **FLog->ShowModal()** добавить оператор:

```
if (FPSW->ShowModal() != 6)
{
    ShowMessage("Ваш пароль неверный");
    Close();
}
else
{
    ShowMessage("Ваш пароль '" + FPSW->EPSW->Text + "'");
    delete FPSW;
}
```

Этот оператор анализирует значение свойства **ModalResult** формы запроса пароля. Значение этого свойства возвращает функция **FPSW->ShowModal()**. Если результат не равен 6, то был введен неправильный пароль. Тогда главная форма, а с ней вместе и приложение, закрываются методом **Close**. При правильном пароле можно продолжать работу приложения. Оператор **ShowMessage** введен просто для того, чтобы показать, как можно использовать свойство другой формы — в данном случае текст, введенный пользователем в качестве пароля. В реальном приложении по этому паролю можно было бы определить уровень доступа пользователя к конфиденциальной информации. Затем следует уничтожение формы запроса пароля операцией **delete** (см. главу 12 раздел 12.9). Это необходимо сделать, чтобы освободить память. Сама по себе эта форма в момент ее закрытия не уничтожается, поскольку по умолчанию закрыть форму — значит сделать ее невидимой. Уничтожать форму до этого момента было нельзя, так как при этом уничтожилась бы содержащаяся в ней информация — введенный пароль.

На этом разработка нашего приложения закончена. Можете сохранить проект, запустить приложение и посмотреть, как оно работает.

Описанный выше способ управления формой запроса пароля не является оптимальным. Он просто призван был показать, как можно обрабатывать величину **ModalResult**, возвращаемую методом **ShowModal**. Но то же самое можно было бы сделать и проще. В обработчике события **OnClose** формы **FPSW** можно было бы написать оператор:

```
if (Edit1->Text != "1") Application->Terminate();
```

При неверном пароле этот оператор завершает работу всего приложения методом **Application->Terminate()**. Тогда в главной форме не надо анализировать результат работы пользователя с формой **FPSW**, так как если приложение не закрылось при выполнении оператора **ShowModal**, то значит пароль введен правильный. Поэтому операторы в главной форме тоже упрощаются:

```
FPSW->ShowModal();
ShowMessage("Ваш пароль '" + FPSW->EPSW->Text + "'");
delete FPSW;
```

Проведите в вашем приложении соответствующие замены операторов и убедитесь, что приложение и в этом случае работает правильно.

4.5.4 Управление формами в приложениях с интерфейсом множества документов (приложениях MDI)

Типичным приложением MDI является привычный всем Word. В приложении MDI имеется родительская (первичная) форма и ряд дочерних форм (называемых также формами *документов*). Окна документов могут создаваться самим пользова-

телем в процессе выполнения приложения с помощью команд типа **Окно | Новое**. Число дочерних окон заранее неизвестно — пользователь может создать их столько, сколько ему потребуется. Окна документов располагаются в клиентской области родительской формы. Поэтому чаще всего целесообразно в родительской форме ограничиваться только главным меню, инструментальными панелями и, если необходимо, панелью состояния, оставляя все остальное место в окне для окон дочерних форм. При этом обычно окно родительской формы в исходном состоянии разворачивают на весь экран.

Из родительской формы можно управлять дочерними формами.

Дочернюю форму нельзя уничтожить, пока не уничтожена родительская форма.

Требования, которые надо учитывать при разработке приложений MDI, подробно рассмотрены в разделе 4.1.2.

Для создания приложения MDI необходимо спроектировать родительскую и дочернюю формы. В родительской форме свойство **FormStyle** устанавливается в **fsMDIForm**, а в дочерней — в **fsMDIChild**. Поскольку дочерние окна будет создавать сам пользователь в процессе выполнения приложения, дочернюю форму необходимо исключить из числа создаваемых автоматически (в разделе 4.5.1 рассказывалось, как это сделать с помощью окна **Опций** проекта).

Рассмотрим теперь, как можно сделать обработчик команды, по которой пользователь задает в родительском окне создание нового окна документов — нового экземпляра дочерней формы. Этот обработчик может иметь вид:

```
класс_дочерней_формы * имя = new класс_дочерней_формы (Application);
if (!имя) return;
операторы_настройки, если они нужны
имя->Show();
```

Первый оператор процедуры создает методом **new** объект дочерней формы и указывает на него с некоторым произвольным временным именем. Далее могут следовать какие-то операторы настройки нового дочернего окна. Например, новому окну надо присвоить какой-то уникальный заголовок (свойство **Caption** дочерней формы), чтобы пользователь мог отличать друг от друга окна документов. Последний оператор процедуры делает видимым вновь созданное окно.

Пусть, например, вы создали в модуле **UMain** родительскую форму, содержащую раздел меню **Окно | Новое**, и создали в модуле **UDoc** дочернюю форму с именем **FDoc**, имеющую тип **TFDoc** (посмотреть для контроля имя и тип дочерней формы вы можете в верхнем выпадающем списке **Инспектора Объектов**, выделив интересующую вас форму, или в модуле, посмотрев автоматически создаваемый **C++Builder** оператор, объявляющий переменную формы и расположенный сразу после директив препроцессора).

Тогда в модуль родительской формы вы должны вставить директиву препроцессора, подключающую заголовочный файл дочерней формы **UDoc** (см. разделы 4.5.1 и 4.5.3). А в обработчике события, связанного с выбором пользователем раздела меню **Окно | Новое**, можно написать операторы:

```
TFDoc *TF = new TFDoc(Application);
if (! TF) return;
...
TF->Show();
```

В родительской форме имеется ряд свойств, позволяющих управлять дочерними окнами. Все они доступны только для чтения и только во время выполнения.

Свойство **MDIChildCount** определяет количество открытых дочерних окон. Приведем оператор, который можно вставить в предыдущий пример для задания уникального имени вновь созданного окна **TF**:

```
TF->Caption = "Документ " + IntToStr(MDIChildCount);
```

Свойство **MDIChildren**[int i] дает доступ к i-му окну (окна индексируются в порядке их создания, последнее созданное окно имеет индекс 0). Однако, во время работы пользователя с окнами индексация может измениться. Поэтому можно рекомендовать использовать индексы только в циклах при проведении некоторых операций сразу со всеми дочерними окнами. Следующий пример показывает процедуру, с помощью которой из родительской формы **Form1** можно закрыть (свернуть) все дочерние окна, начиная с последнего:

```
for (int i = MDIChildCount-1; i >= 0; i-)  
    MDIChildren[i]->Close();
```

В момент создания окон документов они автоматически располагаются каскадом в клиентской области родительской формы. При этом, если размера клиентской области не хватает для размещения дочерних окон, размеры последних автоматически уменьшаются. Имеется ряд методов родительской формы, упорядочивающих размещение дочерних окон. Метод **Cascade** располагает все открытые (не свернутые) окна каскадом. Метод **Tile** располагает окна мозаикой. При этом учитывается свойство родительской формы **TileMode**. Если оно равно **tbVertical**, то упорядочивание производится по вертикали, а если **TileMode** равно **tbHorizontal**, то упорядочивание производится по горизонтали. Метод **ArrangeIcons** упорядочивает расположение пиктограмм свернутых окон.

Отдельно надо упомянуть об объединении главных меню родительской и дочерних форм. Обычно обе эти формы имеют главные меню, но они различны. Например, родительская форма может иметь меню **Окно**, а дочерняя форма — меню **Файл** и **Правка**. Меню дочерних форм не должно появляться в окнах документов, а должно всегда встраиваться в главное меню родительской формы. Поэтому свойство **AutoMerge** компонента типа **TMainMenu** на приложения MDI не влияет: встраивание меню происходит независимо от значения этого свойства. А места, на которые встраиваются разделы меню дочерней формы, определяются значениями свойства **GroupIndex** каждого раздела меню так же, как это имеет место в обычных многооконных приложениях при задании свойства **AutoMerge** равным **true** (см. раздел 3.6.1 главы 3).

4.5.5 Пример приложения с интерфейсом множества документов — простой многооконный редактор

Рассмотрим проектирование простого приложения MDI. Пусть мы хотим создать многооконный редактор, в каждое окно которого можно загрузить содержимое заданного входного текстового файла, что-то в нем изменить и сохранить текст в заданном выходном файле (рис. 4.16).

Обратите внимание на следующие требования к интерфейсу MDI, проиллюстрированные на рис. 4.16.

Хороший стиль программирования

Если новое окно открывается без загрузки текста из файла, заголовок окна должен содержать уникальное имя типа «Документ ...» или «Окно ...», где вместо многоточия должен быть неповторяющийся номер окна. Если в окно загружается текст из файла, то заголовок окна должен содержать имя загруженного файла.

Хороший стиль программирования

Когда окно документа разворачивается, его имя должно включаться в заголовок главного окна (рис. 4.16 в). К счастью, C++Builder производит эту операцию автоматически.

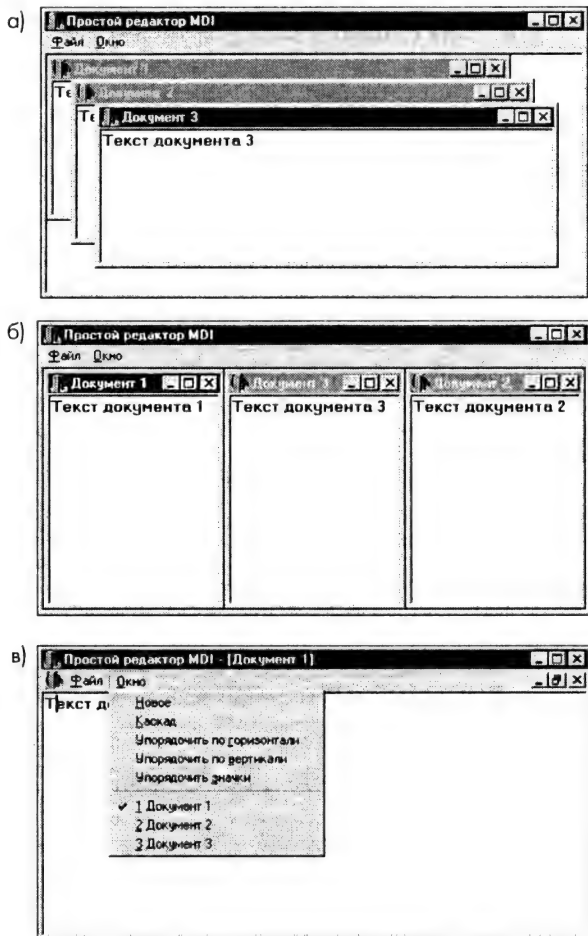
Хороший стиль программирования

Меню Окно должно завершаться разделами, содержащими список открытых окон (см. рис. 4.16 в). Как это сделать — рассказано в главе 3 в разделе 3.6.1 и будет повторено в рассматриваемом примере.

Построим дочерние окна с использованием компонентов **RichEdit**, но для простоты не будем тратить время на разработку сервиса, необходимого в реальных редакторах. Более серьезный пример имеется на прилагаемом к книге диске.

Рис. 4.16

Пример простого приложения MDI: многооконный редактор в режимах упорядочивания окон каскадом (а), по вертикали (б), с развернутым окном документа (в)



Начнем с построения формы окна документа.

1. Откройте в C++Builder новое приложение.
2. Назовите открывшуюся форму **FDoc**.
3. Разместите на форме компонент **RichEdit1** типа **TRichEdit**. Его свойство **Align** задайте равным **alClient**, чтобы окно редактирования заняло всю площадь окна. Сотрите текст, появившийся в **RichEdit1** (свойство **Lines**).
4. Разместите на форме по одному компоненту типов **TOpenDialog** и **TSaveDialog** (см. раздел 3.8.2). Задайте в обоих диалогах свойства **DefaultExt** равными **rtf**, а в свойства **Filter** занесите строку «текстовые файлы» с шаблоном «*.rtf;*.txt» и строку «все файлы» с шаблоном «*.*».

5. Разместите на форме компонент типа **MainMenu**. Создайте в нем раздел Файл (присвойте его свойству **Name** значение **MFile**) с подразделами Открыть (**Name = MOpen**) и Сохранить как (**Name = MSave**).
6. Теперь напишите обработчики событий для введенных подразделов меню. Для раздела **MOpen** обработчик может иметь вид:

```
if (OpenDialog1->Execute())  
{  
    RichEdit1->Clear();  
    RichEdit1->Lines->LoadFromFile (OpenDialog1->FileName);  
}
```

Для раздела **MSave** обработчик может иметь вид:

```
if (SaveDialog1->Execute())  
    RichEdit1->Lines->SaveToFile (SaveDialog1->FileName);
```

7. Сохраните проект, дав спроектированному модулю имя **UDoc**.
Простенький редактор (пока однооконный) построен. Можете скомпилировать его и проверить в работе.
Теперь давайте спроектируем родительскую форму и превратим наш однооконный редактор в многооконный.
8. Измените свойство **FormStyle** созданной вами ранее формы на **fsMDIChild**.
9. Откройте в вашем проекте новую форму (File | New Form). Назовите ее **FMDI**. Сохраните ее модуль с именем **UMDI** (File | Save As).
10. Измените свойство **FormStyle** новой формы на **fsMDIForm**. Задайте свойство **WindowState** равным **wsMaximized**, чтобы окно этой формы предъявлялось пользователю в первый момент развернутым на весь экран. Введите (вручную или командой File | Include Unit Hdr) директиву препроцессора **#include**, ссылающуюся на заголовочный файл **UMDI.h** модуля **UMDI**. Иначе вы не сможете открывать дочерние формы и управлять ими.
11. Выполните команду Project | Options и в открывшемся окне на странице Forms переведите дочернюю форму **FDoc** из списка автоматически создаваемых в список доступных форм. При этом, как вы сможете убедиться по выпадающему списку вверху окна, главной станет родительская форма **UMDI** — единственная, оставшаяся в списке автоматически создаваемых форм.
12. Введите на форму **UMDI** компонент типа **MainMenu**. Сформируйте в нем раздел меню Окно (имя **Name = MWind**) с подразделами Новое (**Name = MNew**), Каскад (**Name = MCascade**), Упорядочить по горизонтали (**Name = MHor**), Упорядочить по вертикали (**Name = MVert**), Упорядочить значки (**Name = MIcons**). Свойство формы **WindowMenu** установите равным **MWind**. Это обеспечит появление внизу меню Окно разделов, содержащих список открытых окон.

Чтобы введенные разделы меню не заменялись во время выполнения разделами меню дочернего окна, необходимо задать им все значения свойства **GroupIndex**, отличные от тех, которые имеют разделы меню дочерней формы. Если этого не сделать, то в процессе выполнения при открытии первого же дочернего окна меню Окно заменится на меню дочерней формы Файл. Следовательно, дальнейшее управление окнами будет потеряно. Это произойдет потому, что по умолчанию значения свойства **GroupIndex** для всех разделов всех меню равно 0.

Если мы хотим, чтобы в процессе выполнения приложения меню дочерней формы Файл встраивалось перед меню Окно, необходимо всем разделам меню Окно присвоить значение **GroupIndex**, большее, чем 0, соответствующий разделам дочернего меню.

13. Во всех введенных разделах меню **Окно** установите свойство **GroupIndex** равным 1.
14. Запишите обработчик события **OnClick** для раздела меню **Новое (MNew)**. Этот обработчик может иметь вид, уже рассмотренный в предыдущем разделе:

```
void __fastcall TFMDI::MNewClick(TObject *Sender)
{
    TFDoc* TF = new TFDoc(Application);
    if (!TF) return;
    TF->Caption = "Документ " + IntToStr(MDICHildCount);
    TF->Show();
}
```

15. Внесите операторы в обработчики событий **OnClick** для остальных разделов меню. Для раздела **Каскад**:

```
Cascade();
```

Для раздела **Упорядочить по горизонтали**:

```
TileMode = tbHorizontal;
Tile();
```

Для раздела **Упорядочить по вертикали**:

```
TileMode = tbVertical;
Tile();
```

Для раздела **Упорядочить значки**:

```
ArrangeIcons();
```

На этом проектирование многооконного редактора завершено. Можете сохранить проект и опробовать приложение в работе (см. рис. 4.16). Посмотрите, как ведет себя приложение при создании нового окна документа, если размеры родительского окна не достаточно велики. Проверьте, что было бы, если бы вы не изменили свойство **GroupIndex** в меню родительского окна или если форма документов имела бы свойство **FormStyle** равное **fsNormal**. Вообще поэкспериментируйте с этим приложением, чтобы уяснить все особенности приложений MDI.

4.5.6 Объект **Screen** и приложения, работающие с несколькими мониторами

В приложении C++Builder автоматически создается глобальный объект **Screen** (экран) типа **TScreen**, свойства которого определяются из информации Windows о мониторе, на котором запускается приложение. Вы можете в любом приложении использовать, например, такие свойства объекта **Screen**, как **Height** — высота экрана и **Width** — его ширина. Это, в частности, может потребоваться, если вы задаете значение свойства своих форм **Position** таким, что размеры форм остаются постоянными. Так как вы используете в процессе проектирования один тип монитора, а приложение в дальнейшем может работать на мониторе другого типа, то не исключено, например, что ваша форма не поместится на экране или наоборот — будет слишком маленького размера для данного монитора. Чтобы избежать этих неприятностей, можно автоматически масштабировать свою форму, вводя, например, в обработчик ее события **OnCreate** код:

```
Width = Screen->Width / 2;
Height = Screen->Height / 2;
```

Этот код задает размеры формы равными половине соответствующих размеров экрана.

Разрешающую способность экрана можно определить, воспользовавшись свойством **PixelsPerInch**, указывающим количество пикселей экрана на дюйм в вертикальном направлении. Это справедливо именно для вертикального направления, поскольку во многих мониторах коэффициенты масштабирования по горизонтали и вертикали различаются.

Screen имеет свойство **Forms[int Index]**, содержащее список форм, отображаемых в данный момент на экране, и свойство **FormCount**, отражающее количество таких форм. Вы можете использовать это свойство, например, для того, чтобы гарантировать, что на данном типе монитора размеры ни одной формы не превысят размеров экрана. Соответствующий код может выглядеть так:

```
for (int i = 0; i < Screen->FormCount; i++)
{
    if (Screen->Forms[i]->Height > Screen->Height)
        Screen->Forms[i]->Height = Screen->Height - 100;
    if (Screen->Forms[i]->Width > Screen->Width)
        Screen->Forms[i]->Width = Screen->Width - 100;
}
```

Размеры форм, превышающие размер экрана, урезаются этим кодом с запасом в 100 пикселей.

В приведенных примерах надо, конечно, предусмотреть, чтобы при изменении размеров формы адекватно изменялось и расположение компонентов на ее поверхности. Этот вопрос подробно рассмотрен в разделе 4.2.

Так же, как к формам, можно получить доступ и к модулям данных (см. раздел 9.11.9). Свойство **DataModules[int Index]** содержит список всех существующих на текущий момент модулей данных приложения, а параметр **DataModuleCount** указывает число таких модулей. Эти параметры можно использовать в совокупности для поиска по всем модулям данных приложения.

Еще одно полезное свойство объекта **Screen** — **Fonts** (шрифты). Это свойство типа **TStrings** содержит список шрифтов, доступных на данном компьютере (свойство только для чтения). Его можно использовать в приложении, чтобы проверять, имеется ли на компьютере тот или иной шрифт, используемый в приложении. Если нет — то можно или дать пользователю соответствующее предупреждение, или сменить шрифт в приложении на один из доступных, или дать пользователю возможность самому выбрать соответствующий шрифт. Например, вы можете поместить в вашем приложении компонент списка **TComboBox** и при событии формы **OnCreate** загрузить его доступными шрифтами с помощью операторов:

```
ComboBox1->Items = Screen->Fonts;
ComboBox1->ItemIndex = 0;
```

Тогда в нужный момент пользователь может выбрать подходящий шрифт из списка, а для того, чтобы этот шрифт использовался, например, для текста в компоненте **RichEdit1**, в обработчик события **OnClick** или **OnChange** списка вставьте операторы:

```
RichEdit1->SelAttributes->Name =
    ComboBox1->Items->Strings[ComboBox1->ItemIndex];
RichEdit1->SetFocus();
```

Если хотите использовать выбранный шрифт для всех компонентов формы, в которых свойство **ParentFont** установлено в **true**, то приведенный выше оператор должен иметь вид:

```
Font->Name = ComboBox1->Items->Strings[ComboBox1->ItemIndex];
```

Свойство **Cursor** объекта **Screen** определяет вид курсора. Если это свойство равно **crDefault**, то вид курсора при перемещении над компонентами определяется установленными в них свойствами **Cursor**. Но если свойство **Cursor** объекта **Screen** отлично от **crDefault**, то соответствующие свойства компонентов отменяются и

курсор имеет глобальный вид, заданный в **Screen**. Этим можно воспользоваться для такой частой задачи, как изменение курсора на форму «песочные часы» во время выполнения каких-то длинных операций. Подобное изменение формы курсора можно оформить следующим образом:

```
Screen->Cursor = crHourGlass;
try
{
    // выполнение требуемых длинных операций
}
catch (...)
{
    Screen->Cursor = crDefault; // восстановление курсора
    throw;
}
Screen->Cursor = crDefault;
```

При успешном или аварийном (см. раздел 12.10) окончании длинных операций курсор в любом случае возвращается в значение по умолчанию.

Если в приложении в какие-то отрезки времени используется отличный от **crDefault** глобальный вид курсора, то приведенный код можно изменить, чтобы по окончании длинных операций восстановить прежнее глобальное значение:

```
TCursor Save_Cursor = Screen->Cursor;
Screen->Cursor = crHourGlass;
try
{
    // выполнение требуемых длинных операций
}
catch (...)
{
    Screen->Cursor = Save_Cursor;
    throw;
}
Screen->Cursor = Save_Cursor;
```

Имеется также свойство **Cursors[I]**, которое представляет собой список доступных приложению курсоров. Вы можете создать и использовать также свой собственный курсор. Создается он и включается в ресурс приложения встроенным в C++Builder Редактором Изображений (Image Editor). Как работать с этим редактором при создании курсора поясняется в разделе 5.1.2.4 главы 5. А регистрируется созданный вами курсор с помощью функции **LoadCursor**. Сделать это можно следующим образом.

Пусть, например, вы создали свой курсор и включили его в ресурс приложения под именем **NEWCURSOR**. Тогда в своем приложении вы можете ввести глобальную константу, обозначающую ваш курсор. Например:

```
const crMyCursor = 1;
```

Значение этой константы может лежать в пределах от -32768 до 32767. Но важно, чтобы она не совпадала с предопределенными значениями стандартных курсоров, лежащими в диапазоне от 0 до -21.

В обработчике события **OnCreate** формы вы можете ввести оператор, регистрирующий ваш курсор в свойстве **Cursors**:

```
Screen->Cursors[crMyCursor]=LoadCursor(HInstance, "NEWCURSOR");
```

Обратите внимание на то, что имя курсора пишется обязательно заглавными буквами, так как именно так хранятся имена курсоров в ресурсах приложения.

В нужный момент вы можете установить этот курсор в качестве глобального оператором

```
Screen->Cursor = crMyCursor;
```


а затем восстановить значение глобального курсора оператором

```
Screen->Cursor = crDefault;
```

Вы можете использовать ваш зарегистрированный курсор и как локальный, например, для панели **Panel1** оператором

```
Panel1->Cursor = (TCursor)crMyCursor;
```

Пример использования различных курсоров вы найдете в разделе 5.1.6 главы 5.

С помощью **Screen** можно получить доступ к активной в текущий момент форме вашего приложения через свойство **ActiveForm**. Если в данный момент пользователь переключился с вашего приложения на какое-то другое и, следовательно, ни одна форма вашего приложения не активна, то **ActiveForm** указывает на форму, которая станет активной, когда пользователь вернется к вашему приложению. В момент переключения фокуса с одной вашей формы на другую, генерируется событие **OnActiveFormChange**.

Аналогично с помощью свойства **ActiveControl** можно получить доступ к активному в данный момент оконному компоненту на активной форме. При смене фокуса генерируется событие **OnActiveControlChange**.

Начиная с C++Builder 4 предусмотрена возможность разработки мультискринных приложений, работающих одновременно с множеством мониторов. При этом приложение может решать, какие формы и диалоги надо отображать на том или ином мониторе. Свойства различных мониторов, используемых в таком приложении, можно найти с помощью свойства **Screen->Monitors[I]**, где **I** — индекс монитора. Индекс 0 относится к первичному монитору. Свойство **Screen->Monitors[I]** является списком объектов типа **TMonitor**, содержащих информацию о конкретных мониторах.

Среди свойств объектов типа **TMonitor** имеются **Height** — высота и **Width** — ширина экрана монитора. Кроме того имеются свойства **Left** и **Top**. Эти свойства означают следующее. Все доступное экранное пространство можно представить себе разбитым на экраны отдельных мониторов, размещающихся слева направо и сверху вниз. Соответственно свойства **Left** и **Top** определяют координаты левого верхнего угла экрана монитора в этом логическом экранном пространстве. Объекты типа **TMonitor** имеют также свойство **MonitorNum** — номер монитора, представляющий собой его индекс в свойстве **Screen->Monitors[I]**.

Для управления тем, на каком мониторе должна появляться та или иная форма, служит свойство формы **DefaultMonitor**. Это свойство может принимать значения:

dmDesktop	не предпринимается попыток разместить форму на конкретном мониторе
dmPrimary	форма размещается на первом мониторе в списке Screen->Monitors
dmMainForm	форма появляется на том мониторе, на котором размещена главная форма
dmActiveForm	форма появляется на том мониторе, на котором размещена текущая активная форма

4.6 Печать в C++Builder

Печать в C++Builder может осуществляться различными способами. В данном разделе обсуждаются простые способы печати текстов и изображений. Составление и печать сложных отчетов рассмотрены в главе 11 в разделе 11.2.

4.6.1 Печать форм методом Print

Формы в C++Builder имеют метод **Print**, который печатает клиентскую область формы. При этом полоса заголовка формы и полоса главного меню не печатаются. Таким образом, можно включить в приложение форму, в которой пользователь во время выполнения размещает необходимые для печати результаты: тексты и изображения. Если имя этой формы **Form2**, то ее печать может выполняться оператором

```
Form2->Print();
```

Свойство формы **PrintScale** определяет опции масштабирования изображения при печати. Возможные значения **PrintScale**:

poNone	Масштабирование не используется. Размер изображения может изменяться в зависимости от используемого принтера
poProportional	Делается попытка напечатать изображение формы того же размера, который виден на экране
poPrintToFit	Увеличивает или уменьшает размер изображения, подгоняя его под размер страницы, заданный при установке принтера. Это значение принято по умолчанию

4.6.2 Методы компонентов, обеспечивающие печать

Ряд компонентов в C++Builder имеют методы, обеспечивающие печать хранящихся в них данных. Например, компонент **RichEdit** имеет метод **Print**, позволяющий печатать в обогащенном формате текст, хранящийся в компоненте. В этот метод передается единственный параметр типа строки, назначение которого заключается только в том, что при просмотре в Windows очереди печатаемых заданий принтера эта строка появляется как имя задания. Например, оператор

```
RichEdit1->Print("Printing of RichEdit1");
```

обеспечивает печать текста компонента **RichEdit1**, причем задание на печать получает имя «Printing of RichEdit1».

Печать воспроизводит все заданные особенности форматирования. Перенос строк и разбиение текста на страницы производится автоматически. Длина строк никак не связана с размерами компонента **RichEdit**, содержащего этот текст.

Печатью через **RichEdit** можно воспользоваться и для печати файлов документов в текстовом формате или в формате RTF. Для этого надо последовательно выполнить оператор загрузки файла в компонент и оператор печати загруженного текста. Например:

```
RichEdit1->Lines->LoadFromFile("Test.txt");
RichEdit1->Print("печать файла Test.txt");
```

или

```
RichEdit1->Lines->LoadFromFile("Test.rtf");
RichEdit1->Print("печать файла Test.rtf");
```

Компонент **Chart**, используемый для отображения графиков и диаграмм (см. раздел 3.4.6), также имеет метод **Print**, обеспечивающий печать. Предварительно может быть выполнен метод **PrintPortrait**, задающий книжную (вертикальную) ориентацию бумаги, или метод **PrintLandscape**, задающий альбомную (горизонтальную) ориентацию. Масштабировать размер печатаемого графика можно, вызвав предварительно метод **PrintRect**,

```
procedure PrintRect ( Const R : TRect ) ;
```

в котором параметр **R** определяет размер области принтера, в которой осуществляется печать.

Компонент **Chartfx** (см. раздел 3.4.7) имеет быструю кнопку печати (пятая слева в инструментальной панели рис. 3.25), с помощью которой пользователь в любой момент может напечатать текущий график или диаграмму.

4.6.3 Печать средствами офисных приложений Windows с помощью функции ShellExecute и обращения к серверам COM

Для печати файлов средствами стандартных офисных приложений Windows можно использовать функцию **ShellExecute**. Подробно об этой функции см. раздел 6.1.4. А здесь мы коротко рассмотрим технологию такой печати без каких-либо дополнительных пояснений.

Чтобы воспользоваться функцией **ShellExecute**, надо ввести в модуль директиву препроцессора

```
#include "ShellApi.h"
```

которая подключает заголовочный файл **ShellApi.h**, содержащий объявление функции **ShellExecute** и некоторых других функций Windows API. Функция **ShellExecute** при соответствующем задании ее параметров ищет по расширению заданного для печати файла соответствующую ему системную программу Windows, и, если находит, то осуществляет печать. Например, обычно Windows настроен так, что файлам с расширением **.txt** соответствует программа Notepad, а файлам с расширением **.doc** — Word. В этом случае выполнение оператора

```
ShellExecute(Handle, "print", "Test.txt", NULL, NULL, SW_HIDE);
```

вызовет печать файла с именем **test.txt** средствами программы Notepad, а оператор

```
ShellExecute(Handle, "print", "Test.doc", NULL, NULL, SW_HIDE);
```

вызовет печать файла с именем **test.doc** средствами программы Word.

Этот способ печати можно использовать как для распечатки заранее созданных файлов, так и для распечатки файлов, созданных во время выполнения приложения методами **SaveToFile**, имеющимися у многих компонентов.

Имеется также возможность печатать тексты и графику средствами стандартного редактора Windows Word или средствами Excel. Для этого в C++Builder 5 имеются компоненты — серверы COM, позволяющие сначала создать документ соответствующего приложения Windows, а затем его напечатать. Эти возможности рассмотрены в главе 6 в разделе 6.4.4.

4.6.4 Печать с помощью объекта Printer

В C++Builder имеется класс печатающих объектов **TPrinter**, который обеспечивает печать текстов, изображений и других объектов, расположенных на его канве — **Canvas**. Свойства канвы подробно рассмотрены в разделе 5.1.3 и здесь мы не будем на них останавливаться. Достаточно знать, что на канве могут размещаться различные изображения и текст.

Класс объектов **TPrinter** объявлен в модуле **Printers**. Поэтому для работы с этим классом надо включить в текст директиву препроцессора

```
#include <Printers.hpp>
```

Рассмотрим некоторые свойства и методы объекта типа **TPrinter**.

Свойство, метод	Описание
Canvas	Канва Canvas — место в памяти, в котором формируется страница или документ перед печатью. Canvas обладает рядом свойств, включая Pen (перо) и Brush (кисть), которые позволяют вам делать рисунки и помещать на них текст. Подробное описание канвы и методов работы с ней вы найдете в разделе 5.1.3
TextOut	Метод канвы, который позволяет посылать в нее текст
Draw	Метод канвы, который позволяет посылать в нее изображение
BeginDoc	Используется для начала задания печати
EndDoc	Используется для окончания задания печати. Фактическая печать происходит только при вызове EndDoc
PageHeight	Высота страницы в пикселях
PageWidth	Ширина страницы в пикселях
NewPage	Принудительно начинает новую страницу на принтере
PageNumber	Возвращает текущий номер печатаемой страницы

Предположим, вы хотите напечатать текст и изображение, используя печатающий объект. Изображение размещено на форме в компоненте **Image1**. Вы можете осуществить эту печать следующим кодом:

```
TPrinter *Prntr = Printer();
Prntr->Canvas->Font->Size = 12;
Prntr->BeginDoc();
Prntr->Canvas->TextOut(10,10,"Я печатаю через объект Printer");
Prntr->Canvas->Draw(
    (Prntr->PageWidth - Image1->Picture->Bitmap->Width)/2,
    40, Image1->Picture->Bitmap);
Prntr->EndDoc();
```

Первый оператор этого кода использует функцию **Printer**, которая создает глобальный объект типа **TPrinter**. В том же операторе создается указатель на этот объект **Prntr**.

Следующий оператор задает размер шрифта канвы принтера. Затем функция **BeginDoc** запускает задание на печать. Следующий оператор посылает на канву принтера с помощью метода канвы **TextOut**, начиная с точки с координатами (10, 10), текст «Я печатаю через объект Printer». Следующий оператор методом **Draw** рисует на канве принтера изображение. При этом изображение выравнивается по горизонтали на середину страницы. Координата верхней стороны изображения задается равной 40.

В заключение метод **EndDoc** вызывает печать текста и изображения и останавливает задание на печать.

Печатающий объект **Printer** не производит автоматического переноса строк и разбиения текста на страницы. Поэтому печать длинных текстов с помощью объекта **Printer** требует достаточно сложного программирования. Проще это делать описанными ранее способами или с помощью системы **QuickReport**, которая описана в разделе 11.2.

4.7 Развертывание приложений

4.7.1 Оформление завершенного проекта

4.7.1.1 Задание учетной информации о версии завершенного приложения

Если вы делаете приложение, которое в дальнейшем будет распространяться среди пользователей, вы обязаны снабдить его положенной учетной информацией о версии данного продукта, о его назначении и т.д. Впрочем, и для самого себя полезно указывать подобную информацию, чтобы не запутаться в многочисленных версиях разрабатываемого приложения.

Для задания или просмотра этой информации надо выполнить команду Project Options и в открывшемся окне опций проекта перейти на страницу Version Info (рис. 4.17). Указанная на данной странице информация заносится в выполняемый файл и становится доступна пользователю, когда он щелкает правой кнопкой мыши на пиктограмме вашего приложения и выбирает команду Свойства. Тогда в открывшемся диалоговом окне на странице Версия пользователь может увидеть информацию о вашем приложении (см. рис. 4.18).

Рис. 4.17
Страница Version Info окна Project Options

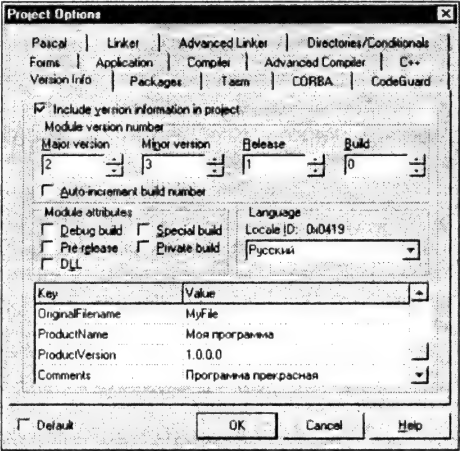
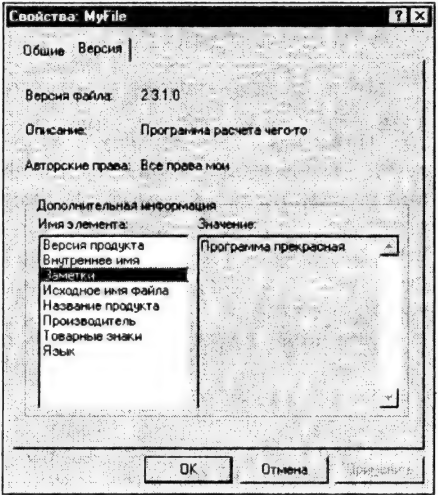


Рис. 4.18
Информация о приложении, которую видит пользователь, работая в Windows



Основная опция страницы Version Info — Include version information in project. Если она не установлена, то все окна страницы доступны только для чтения. Это позволяет просто посмотреть информацию о версии вашего прошлого приложения, если, конечно, вы позаботились о том, чтобы ввести ее в приложение. Если же вы установите опцию Include version information in project, то все окна страницы станут доступны для ввода информации.

Окна группы Module Version Number позволяют вам задать номер версии, состоящий из четырех цифр, разделенных точками. Например, 2.3.1.0. Этот номер пользователь видит в верхней части окна (рис. 4.18). Индикатор Auto-increment build number позволяет автоматизировать изменение последней из этих цифр. Если вы включили этот индикатор, последняя цифра будет увеличиваться на единицу каждый раз, когда вы будете выполнять команду Project | Build All. При других командах компиляции цифра изменяться не будет.

Группа индикаторов Module Attributes указывает назначение версии. Эти индикаторы можно заполнять для себя в чисто информационных целях, например: Debug Build — отладка, Pre-Release — версия не для коммерческого использования, DLL — проект DLL, Special Build — версия получена в стандартном режиме компиляции, Private Build — версия построена не в стандартном режиме компиляции.

Окно Language указывает кодовую страницу системы пользователя, которая нужна для запуска приложения, т.е. указывает язык. Соответственно в окне рис. 4.18 при выделении пользователем строки Язык он увидит язык приложения (константа 0x0419 соответствует русскому языку). Окно Language заполняется автоматически в зависимости от установленной в системе кодовой страницы.

Список Key/Value включает в себя стандартные сведения о программном продукте: CompanyName (Производитель), FileDescription (Описание — в окне рис. 4.18 вторая строка вверху окна), FileVersion (Версия продукта), InternalName (Внутреннее имя), LegalCopyright (Авторские права — в окне рис. 4.18 третья строка вверху окна), LegalTrademarks (Товарные знаки), OriginalFilename (Исходное имя файла), ProductName (Название продукта), ProductVersion (Версия продукта), Comments (Заметки). В этом перечислении в скобках указаны заголовки, которые пользователь видит в окне рис. 4.18. Перемещая курсор в этом окне, пользователь может читать то, что вы занесли в список Key/Value. Вообще говоря, для коммерческого продукта все строки списка, кроме LegalCopyright, LegalTrademarks и Comments, должны заполняться, а эти три строки могут заполняться при необходимости.

4.7.1.2 Интернационализация приложения

Если вы предназначаете свое приложение для международного рынка или хотя бы для международной демонстрации, вам надо позаботиться о его интернационализации. Надо, чтобы в зависимости от того, какой язык установлен в Windows на конкретном компьютере, ваше приложение автоматически разговаривало бы на этом языке.

Начиная с C++Builder 5 подобная интернационализация приложений существенно облегчилась. Правда, для этого надо спроектировать приложение так, чтобы оно поддавалось интернационализации. Для этого надо изолировать все ресурсы, которые должны изменяться в процессе локализации. То есть перенести в файлы .dfm и .res все, что может изменяться при смене языка. Например, все тексты, используемые в приложении, надо перенести в ресурсы с помощью ключевого слова **resourcestring**.

Все далее изложенное вы можете применить к любому из имеющихся у вас приложений. Но лучше давайте разработаем чисто демонстрационное приложение, чтобы на его примере рассмотреть методику интернационализации. Откройте новое приложение, поместите на форму две метки, список **ListBox**, кнопку, компонент **MainMenu**. Можете также поместить на форму полосу состояния **StatusBar** и компонент **ApplicationEvents**, заносящий в эту полосу подсказки **Hint** компонентов.

Задайте какой-нибудь русский текст в заголовке формы (свойство **Caption**), например, «Интернационализация». Занесите какие-то строки в список **ListBox**, сделайте надписи на первой метке и на кнопке. Задайте тексты ярлычков и подсказок компонентов в их свойствах **Hint**. Задайте значения **true** в свойствах **ShowHint** компонентов. Задайте также какие-то разделы меню. Все это может выглядеть так, как показано на рис. 4.19.

Теперь давайте напишем пару операторов, чтобы наше приложение могло что-то делать. В обработчик события **OnHint** компонента **ApplicationEvents** занесите оператор

```
StatusBar1->SimpleText = Application->Hint;
```

Этот оператор, рассмотренный в разделе 4.1.9, отобразит в панели состояния подсказки **Hint**.

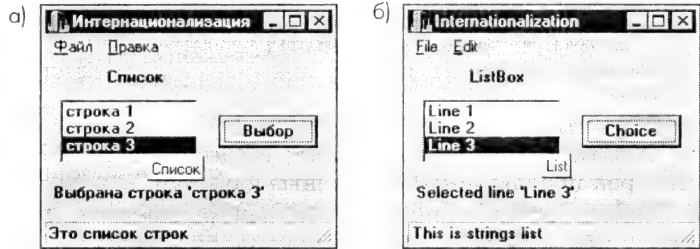
В обработчик события **OnClick** кнопки занесите код

```
if(ListBox1->ItemIndex >= 0)
    Label2->Caption = "Выбрана строка '" +
        ListBox1->Items->Strings[ListBox1->ItemIndex] + "'";
else Label2->Caption = "Выбор не сделан";
```

Этот код отображает в метке **Label2** строку, выбранную пользователем в списке **ListBox1**.

Рис. 4.19

Приложение при установленном русском (а) и английском (б) языке (при русифицированной версии Windows)



Приложение завершено. Сохраните его, выполните и убедитесь, что все работает. Можете, конечно, придумать какое-нибудь свое более интересное приложение. Важно, чтобы в нем были русские надписи на компонентах формы и какие-то русские тексты в коде (в нашем случае это тексты в обработчике события кнопки **OnClick**).

Теперь давайте подготовим ваше приложение к интернационализации. Основное требование к приложению, которое должно разговаривать на разных языках — изоляция ресурсов. В самом выполняемом модуле никаких текстов и никаких изображений, которые должны изменяться в зависимости от языка, не должно быть. Все, что может подвергаться изменениям, должно быть вынесено в динамически присоединяемые библиотеки. Тогда приложение в зависимости от локализации Windows, установленной на конкретном компьютере, сможет подключать соответствующую этой локализации библиотеку.

C++Builder 5 обеспечивает изоляцию файлов изображений форм **.dfm**, и файлов ресурсов **.rc**, в которых могут размещаться тексты. Следовательно, обо всех текстах, отображаемых в компонентах формы, заботиться не надо. Они автоматически будут изолированы вместе с файлом формы. А вот для изоляции текстов, имеющих в приложении, надо принимать специальные меры. Такая изоляция может быть сделана несколькими способами. Наиболее напрашивающийся вариант — ввести в приложение невидимый список **ListBox**, занести в него требуемые строки и извлекать их оттуда, когда потребуется сделать пользователю какое-то сообщение. Это просто, но не очень удобно для программирования, так как придется оперировать не идентификаторами строк, а их индексами в списке. К тому же

такой вариант неэкономичен, так как приходится хранить не просто строки, а целый компонент со всеми его атрибутами.

Другой вариант изоляции строк — занести их в файл ресурсов. Это, в частности, можно сделать, введя в приложение модуль на языке Pascal, в котором с помощью ключевого слова **resourcestring** занести требуемые строки в файл ресурсов **.res**. Рассмотрим подробнее этот вариант на примере нашего приложения.

Текст соответствующего файла на языке Pascal может иметь вид:

```
unit <имя файла>;
interface
resourcestring
// список строк вида идентификатор = строка
implementation
begin
end.
```

Например, в нашем случае требуется перенести в файл ресурсов две строки: «Выбрана строка » и «Выбор не сделан». Дадим им идентификаторы **SSel** и **SNoSel** соответственно. Тогда текст модуля на языке Pascal (дадим ему имя **ures**) должен иметь вид:

```
unit ures;
interface
resourcestring
    SSel = 'Выбрана строка ';;
    SNoSel = 'Выбор не сделан';
implementation
begin
end.
```

Обратите внимание на то, что в языке Pascal, в отличие от языков C и C++, для строк используются одинарные кавычки, а не двойные. Поэтому в текст строки **SSel** введена в конце пара одинарных кавычек, чтобы символ воспринимался именно как кавычка, а не как окончание строки.

Приведенный выше текст можно написать в любом текстовом редакторе, сохранить как «только текст» в файле с именем **ures.pas** и затем включить файл **ures.pas** в проект. Но можно все это сделать в среде C++Builder. Для этого выполните команду **File | New** и в окне Депозитария на странице **New** выберите пиктограмму **Text**. В окне Редактора Кода откроется пустой файл с именем по умолчанию **File1.txt**. Занесите в него приведенный выше текст. Сохраните этот файл (команда **File | Save As**) под именем **ures.pas**, выбрав для этого в окне диалога сохранения файла фильтр **Pascal unit (*.pas)**. Теперь надо включить этот файл в проект. Выполните для этого команду **Project | Add to Project**, в открывшемся диалоговом окне выберите фильтр **Pascal unit (*.pas)** и укажите файл **ures.pas**.

Теперь надо обеспечить доступ из основного модуля программы к файлу **ures.pas**. Это делается обычным образом. Перейдите в окне Редактора Кода в модуль **Unit1**, выполните команду **File | Include Unit Hdr** и в открывшемся диалоговом окне выберите присоединяемый файл **ures.pas**. Взгляните, к чему привело выполнение этой команды. Вы можете увидеть, что в текст модуля введена директива компилятора

```
#include "ures.hpp"
```

Это подключается заголовочный файл **ures.hpp**, который C++Builder автоматически сгенерировал для модуля **ures.pas**. Полезно взглянуть на этот заголовочный модуль. Для этого переведите курсор на имя модуля **ures.hpp** в директиве **#include**, щелкните правой кнопкой мыши и выберите в контекстном меню раздел **Open File at Cursor**. Вы увидите в окне Редактора Кода файл **ures.hpp**. Обратите внимание в нем на следующий раздел:

```
namespace Ures
{
    //- type declarations
    //- var, const, procedure
    extern PACKAGE System::ResourceString _SSel;
    #define Ures_SSel System::LoadResourceString(&Ures::_SSel)
    extern PACKAGE System::ResourceString _SNoSel;
    #define Ures_SNoSel System::LoadResourceString(&Ures::_SNoSel)

} /* namespace Ures */
```

Эти операторы объявляют ключевым словом **namespace** область видимости имен. Введенным вами строкам присвоены имена **_SSel** и **_SNoSel**. Однако тип этих идентификаторов — **ResourceString**. Это структуры, работать с которыми неудобно. Гораздо удобнее работать с идентификаторами **Ures_SSel** и **Ures_SNoSel**, определенными директивами **#define**. Эти директивы определяют макросы, вызывающие функции **LoadResourceString**. Эти функции переводят аргументы типа **ResourceString** в тип **AnsiString**. Следовательно, именно через идентификаторы **Ures_SSel** и **Ures_SNoSel** следует обращаться к строкам, содержащимся в файле ресурсов.

Теперь осталось изменить ранее написанный оператор обработчика щелчка на кнопке следующим текстом:

```
if(ListBox1->ItemIndex >= 0)
    Label2->Caption = Ures_SSel +
        ListBox1->Items->Strings[ListBox1->ItemIndex] + " ";
else Label2->Caption = Ures_SNoSel;
```

В этом операторе тексты, которые были в нем раньше, заменены идентификаторами строк **Ures_SSel** и **Ures_SNoSel**.

Сохраните модернизированный вариант вашего приложения, выполните команду **Project | Build All** и запустите приложение на выполнение. Убедитесь, что все работает нормально.

Предупреждение

Для того, чтобы дальнейшая интернационализация прошла успешно, надо компилировать проект именно командой **Project | Build All**. При других командах нужные файлы не создадутся.

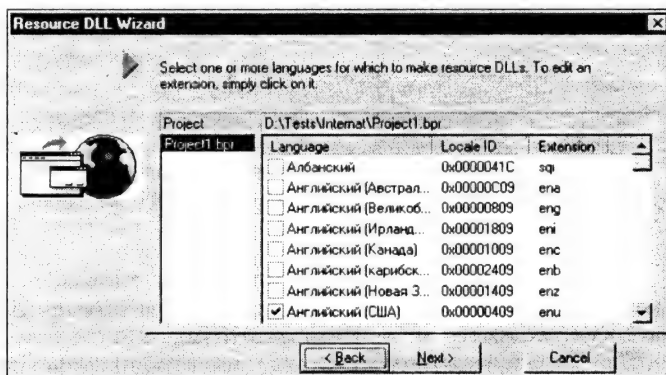
Теперь давайте займемся интернационализацией приложения. Проще всего это сделать, выполнив команду **File | New** и выбрав на странице **New** окна **Депозитария** пиктограмму **Resource DLL wizard** — **Мастер DLL ресурсов**. Предварительно приложение должно быть сохранено на диске и откомпилировано (у нас уже это сделано).

Мастер покажет вам серию экранов, которые, вероятно, не имеет смысла описывать все, так как в них в большинстве случаев вы можете ничего не делать, просто нажимая кнопку **Next** — следующее окно. Остановимся только на тех, в которых вам надо что-то делать. Первое из таких окон, представлено на рис. 4.20. В нем в списке вы должны установить индикаторы тех языков, для которых хотите создать варианты своего приложения. Языков очень много, так что при желании вы можете сделать свое приложение настоящим полиглотом и распространять его по всему миру. Но для начала выберите какой-то один язык, например, английский. Впоследствии вы сможете добавлять к своему приложению другие языки. Русский язык тоже можете указать, хотя это вовсе не обязательно. Он все равно является базовым, т.е. исходным. Впрочем, для большей симметрии дальнейшей работы задайте и его.

Еще одно окно, на которое надо обратить внимание, представлено на рис. 4.21. В нем вы можете кнопкой **Add File** указать какие-то дополнительные файлы, изменяющиеся для разных языков, но не распознаваемых автоматически. Без дополни-

Рис. 4.20

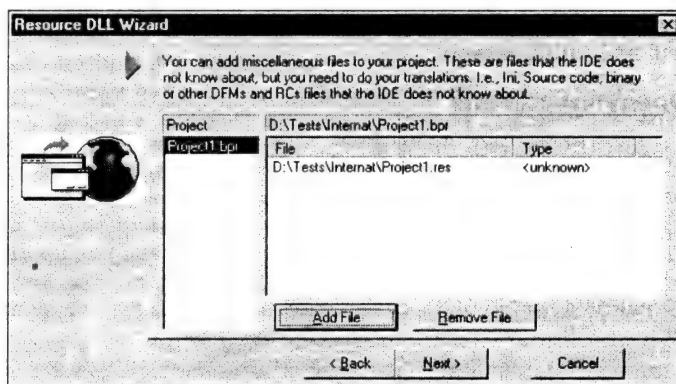
Выбор языков приложения



тельных указаний C++Builder распознает файлы **.dfm** и **.rc**. Но у вас могут быть какие-то дополнительные файлы: ресурсов, текстовые, графические, файлы настройки **.ini** (см. раздел 4.7.3), которые вы захотите иметь различными для разных языков. Например, в нашем приложении могла бы быть предусмотрена загрузка в список **ListBox** какого-то текстового файла, который передается пользователю вместе с приложением. Или приложение могло бы быть снабжено файлом справки **.hlp**, тексты которого, естественно, должны изменяться при смене языка. Подобные файлы следует добавить в окно рис. 4.21. Следует сказать, что большинство типов дополнительных файлов не управляются рассмотренным далее Менеджером Трансляции. C++Builder просто создаст копии этих файлов для каждого языка. А перевод этих копий на тот или иной язык вы должны будете осуществить самостоятельно.

Рис. 4.21

Указание дополнительных файлов, подлежащих трансляции



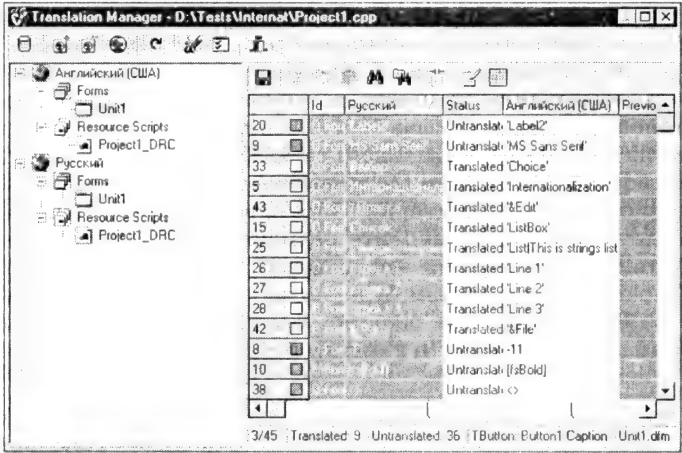
В нашем случае надо добавить файл ресурсов **.res**, в который помещены строки, указанные как **resourcestring**. Нажмите кнопку **Add File**, в открывшемся диалоговом окне выберите шаблон типа файлов **All files (*.*)** и укажите файл ресурсов, имеющий то же имя, что ваш проект, и расширение **.res**.

Пройдя далее через серию окон Мастера DLL ресурсов и нажав в последнем из них кнопку **Finish**, вы увидите окна с сообщениями о том, что отсутствует файл **.rc**, с предложением перекомпилировать проект с опциями, обеспечивающими создание этого файла, и предложением сохранить файл группы проектов. Следует согласиться со всеми сделанными вам предложениями и сохранить файл группы проектов.

Теперь все сделано, чтобы ваш проект смог заговорить на разных языках. Перед вами автоматически откроется окно Менеджера Трансляции (рис. 4.22), в ко-

тором вы можете осуществить трансляцию ресурсов на разные языки. В левой панели окна вы видите дерево ресурсов форм (Forms) и ресурсов проекта (Resource Script) для разных языков. Выделите, например, в английском языке вершину модуля формы Unit1. Вы увидите в правой панели (см. рис. 4.22) свойства формы и ее компонентов, допускающие трансляцию. В первом столбце Id (на рис. 4.22 он показан не полностью) расположены идентификаторы свойств. Во втором столбце приводятся значения этих свойств в исходном русском варианте, следующий столбец указывает, транслировалось или нет данное свойство, а в следующем столбце вы видите значение свойства в английском варианте. Это значение вы можете редактировать. Редакцию можно осуществлять или непосредственно в ячейке таблицы, или можно, выделив ячейку, нажать крайнюю правую быструю кнопку (см. рис. 4.22) и вызвать специальный многострочный редактор. В таблице имеются еще не показанные на рис. 4.22 столбцы, указывающие предыдущую редакцию перевода, дату создания начального варианта и последнего изменения.

Рис. 4.22
Окно Менеджера Трансляции



Вы можете в транслируемом варианте изменить шрифт, множество символов и т.п., чтобы сделать тексты доступными на соответствующем языке. Но главное, что надо сделать — перевести тексты надписей на форме и ее компонентах. Чтобы это было проще осуществить, можно посоветовать предварительно щелкнуть на заголовке столбца Русский. Тогда строки таблицы станут упорядочены в алфавитном порядке значений свойств русского варианта и все надписи на русском языке соберутся вместе, как показано на рис. 4.22.

Аналогичным образом надо заменить строки ресурсов в вершине английского варианта Project1_DRC, содержащие русские надписи «Выбрана строка» и «Выбор не сделан».

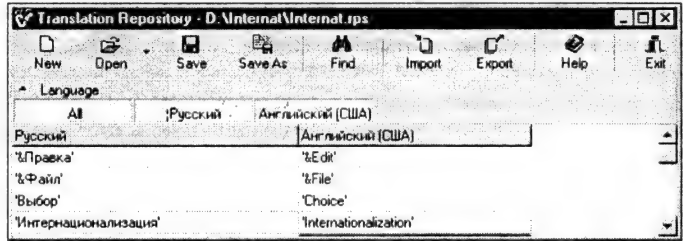
При переходе к вершине Project1_DRC вам будет задан вопрос, хотите ли вы сохранить изменения в вершине Unit1. Аналогичный вопрос о сохранении изменений будет вам задан при выходе из окна Менеджера Трансляции. Конечно на эти вопросы следует ответить положительно. Впрочем, вы можете не дожидаться этих вопросов, а сохранить результаты трансляции, нажав соответствующую быструю кнопку (первая слева в правой группе на рис. 4.22).

Крайняя правая кнопка Actions на рис. 4.22 открывает выпадающее меню с рядом разделов. Это же меню появляется при щелчке правой кнопкой в какой-то строке столбца Русский. Остановимся на одном разделе этого меню — Repository с двумя подразделами Add Strings to Repository и Get Strings from Repository. Первый из этих подразделов сохраняет ваш перевод строки в депозитории переводов. А второй переносит в выделенную строку перевод, хранящийся в депозитории. В сам де-

позиторий (см. рис. 4.23) вы можете попасть, нажав крайнюю левую кнопку в окне рис. 4.22. В нем вы можете хранить переводы различных типовых текстов, встречающихся в приложениях. Так что после интернационализации нескольких приложений у вас будет не так уж много забот с переводом новых текстов.

Рис. 4.23

Окно депозитария трансляции



По рассмотренной выше команде Get Strings from Repository перенос из депозитария осуществляется автоматически, если там нашлось точное соответствие текста, указанного в выделенной строке, причем в депозитарии имеется только один перевод этого текста. Если имеется несколько переводов, то поведение определяется установкой опции Multiple Find Action в окне задания опций трансляции, которое вы можете вызвать командой главного меню C++Builder 5 Tools | Translation Tools Options. Если в опции Multiple Find Action вы установите радиокнопку Skip, то при обнаружении в депозитарии более одного перевода данного текста ничего делаться не будет. Если вы установите радиокнопку Use first, то в этом случае в строку будет вставлен первый из нескольких переводов. Если вы установите радиокнопку Display selection, то вам будет предоставлен выбор из имеющихся переводов.

В том же окне задания опций трансляции имеются также опции группы Resource DLL Wizard. Эти опции означают следующее:

Automatic repository query	Автоматически заполнять строки переводами из депозитария
Automatically compile projects	Автоматически компилировать проект, если необходимо, не запрашивая об этом пользователя
Show Translation Manager after RDW	Автоматически показывать окно Менеджера Трансляции после окончания работы Мастера трансляции

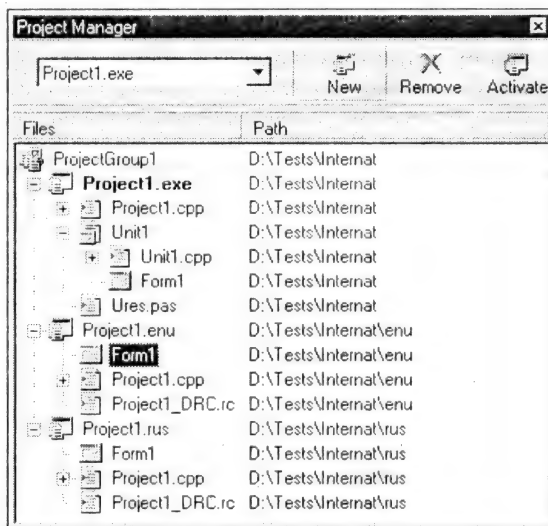
Давайте вернемся к нашему примеру. По окончании работы с Менеджером Трансляции вы увидите, что создалась группа проектов. С ней проще всего работать, вызвав окно Менеджера проектов (команда View | Project Manager), показанное для нашего примера на рис. 4.24. Оно содержит вершину Project1.exe, отображающую сам проект, и вершины Project1.enu и Project1.rus, отображающие соответственно ресурсы английского и русского вариантов. Выбрав вершину Form1 того или иного варианта, вы можете увидеть, как выглядит ваша форма в русском или английском исполнении. Вершины Project1_DRC.rc покажут вам файлы ресурсов обоих вариантов. Но не исправляйте их вручную — это не приведет ни к чему хорошему.

Предупреждение

Не исправляйте вручную в интернационализированном проекте файлы ресурсов. Все исправления делайте только через окно Менеджера Трансляции.

Рис. 4.24

Окно Менеджера Проектов группы оттранслированных вариантов приложения



Сохраните файл группы проектов и попробуйте с ним работать. Активизируйте в окне Менеджера Проектов вершину Project1.exe. Для этого можете сделать на ней двойной щелчок, или выделить ее и нажать кнопку Activate, или нажать в инструментальной полосе среды C++Builder маленькую кнопочку рядом с быстрой кнопкой Run. После этого выполните ваше приложение. Вы увидите ту же картину, что и раньше (рис. 4.19 а).

А теперь выполните команду Project | Languages. Всплывет каскадное меню с разделами Add (добавить новый язык), Remove (удалить один из языков), Set Active (сделать активным один из языков) и Update Resource DLLs (обновить DLL ресурсов).

Предупреждение

Все разделы команды Project | Languages, кроме команды Add, доступны, только если у вас в окне Менеджера Проектов активизирована вершина головного файла проекта (в нашем примере — Project1.exe) или корневая вершина группы.

Команда Add приведет вас в одно из окон Мастера DLL ресурсов, в котором вы можете указать проект и далее добавить к нему новый язык. Команда Remove приведет вас в окно, в котором вы увидите список введенных в проект языков и около каждого языка будут стоять индикаторы. Их вы можете включить у тех языков, которые хотите удалить. Впрочем, удаление коснется только возможности делать соответствующий язык активным. Сами DLL ресурсов на диске сохранятся.

Команда Update Resource DLLs обновляет DLL ресурсов. Выполняйте ее после любых изменений и вообще почаще работайте с ней в случаях возникновения каких-то недоразумений.

Команда Set Active делает активным один из языков. В ответ на эту команду появляется диалоговое окно, в котором вы можете выбрать один из языков для отладки вашего приложения, или выбрать <none>. Последнее означает, что выбор варианта осуществляется автоматически в зависимости от языка, установленного в Windows. Установите активным русский язык (если вы не указывали его как один из языков при работе в окне рис. 4.20, то его не будет в списке команды Set Active). Выполнив проект, вы увидите, что ничего в его работе, конечно, не изменилось. Если же вы сделаете активным английский язык и после этого выполните проект, то увидите (рис. 4.19 б), что ваше приложение заговорило на английском языке.

То, что вы сейчас делали — это только активизация того или иного языка при отладке. К действительному поведению приложения при выполнении его не из среды C++Builder установки языка отношения не имеют. Можете проверить это. Чтобы эксперимент был совсем чистый, перенесите в отдельный каталог файлы **Project1.exe** и **Project1.enu** — английский вариант ресурсов. Закройте C++Builder. Выполните ваш файл (**Project1.exe**) средствами Windows, например, программой «Проводник». Приложение будет разговаривать по-русски. А теперь установите в Windows английский язык. Для этого выполните программу «Панель управления», щелкните в ее окне на пиктограмме Язык и стандарты и на странице Региональные стандарты выберите из выпадающего списка английский язык. Щелкните на ОК. Вам будет предложено перезагрузить компьютер, чтобы ваша установка вступила в силу. Согласитесь на перезагрузку, а после нее опять выполните вне C++Builder ваше приложение. Вы увидите, что оно разговаривает по-английски. А ведь вы ничего с ним не делали! Оно само поняло установленный в системе язык и заговорило на нем. Вот это и есть интернационализация приложения. Порадуйтесь на свое создание, которое стало полиглотом, но не забудьте опять вернуться к нормальным установкам, указав с помощью «Панели управления», что вы все-таки хотите работать с русским языком.

Предупреждение

Не забудьте, что если вы хотите распространять свое интернационализированное приложение, то вместе с выполняемым модулем надо распространять и файлы соответствующих языков (в нашем примере **Project1.enu**).

Мы рассмотрели вариант интернационализации приложения с помощью Мастера DLL ресурсов. Но в дальнейшем вы можете работать с приложением: добавлять, например, новые языки, удалять введенные, изменять перевод и т.п., не прибегая к помощи Мастера. Для этого имеется команда **View | Translation Manager**, вызывающая окно Менеджера Трансляции (рис. 4.22). В нем вы можете не только редактировать свой перевод, но и добавлять новый язык (быстрая кнопка со знаком «+») или удалить имеющийся (быстрая кнопка со знаком «-»). В обоих случаях произойдет автоматическое обращение к тем или иным окнам Мастера DLL ресурсов. Только имейте в виду, что команда **View | Translation Manager** доступна, только если у вас в окне Менеджера Проектов активизирована вершина головного файла проекта или корневая вершина группы.

4.7.1.3 Завершение разработки проекта

По окончании работы над проектом надо провести завершающую компиляцию проекта с соответствующей установкой всех опций. Прежде всего надо решить, будете ли вы использовать поддержку пакетов выполнения (см. раздел 7.6), или будете генерировать автономный выполняемый файл. Если надо генерировать автономный файл, а ранее согласно рекомендациям раздела 7.6 вы отлаживали проект в режиме поддержки пакетов времени выполнения, то необходимо выполнить команду **Project | Options** и в раскрывшемся окне опций проекта на странице **Packages** выключить индикатор **Built with runtime packages**. Если же вы намерены при распространении приложения использовать поддержку пакетов выполнения, то вы должны, пользуясь методикой, рассмотренной в разделе 7.6, отобрать те пакеты, которые будут распространяться вместе с приложением. Обычно полезно проверить полностью комплектации передаваемых пользователю файлов, попробовав выполнить приложение на компьютере, на котором отсутствует C++Builder.

При заключительной компиляции надо также установить опции оптимизации, отключить опции отладки и т.п. Все это делается в окне опций проекта, рассмотренном в разделе 14.2.9. В частности, правильной установкой всех этих опций на странице **Compiler** окна опций проекта способствует кнопка **Release** (см. раз-

дел 14.2.9.2). Но некоторые опции надо устанавливать на страницах Advanced Compiler и C++.

На заключительной стадии разработки проекта надо также решить вопросы установки вашего приложения на компьютерах пользователей. Эти вопросы рассмотрены в следующих разделах.

4.7.2 Установка и настройка приложения

4.7.2.1 Работа с системным реестром

Обсудим коротко вопросы установки вашего приложения на компьютере пользователя. Если это очень простое приложение, то никаких проблем нет — достаточно скопировать выполняемый файл приложения с загрузочной дискеты или диска CD ROM в выделенный для приложения каталог. Если в дальнейшем пользователь решит удалить вашу программу со своего компьютера, ему будет достаточно удалить этот файл.

В более сложных приложениях обычно фигурируют различные файлы настройки, конфигурации и т.д., расположенные к тому же в разных каталогах. В этих случаях установить программу на компьютере и удалить ее, если она стала не нужна, уже гораздо сложнее.

Раньше, в Windows 3.x информация о конфигурации и настройках приложения хранилась в файлах `.ini`. Но для того чтобы упорядочить процессы установки и удаления программ, начиная с Windows 95 и NT Microsoft требует, чтобы вся информация о конфигурации системы хранилась в системном реестре. Реестр (Registry) — это база данных для хранения информации о системной конфигурации аппаратуры, о Windows и о приложениях Windows. Почти все, что в Windows 3.x находилось в файлах `.ini`, перенесено в Windows 95 и NT в реестр. Реестр имеет иерархическую организацию, которая, содержит много уровней ключей, субключей и параметров. Информация хранится в виде иерархического дерева, каждый узел которого называется ключом. Ключ может содержать субключи и значения параметров.

Реестр делит все свои данные на две категории: характеризующие компьютер и характеризующие пользователя. Характеристики компьютера включают в себя все, связанное с техническими средствами, а также с установленными приложениями и их конфигурацией. Характеристики пользователя включают в себя установки по умолчанию для экрана, пользовательские конфигурации, информацию о выбранных пользователем принтерах, установки сети.

Все субключи относятся к шести основным ключам реестра. Четыре из них определяют характеристики компьютера:

Hkey_Local_Machine	Информация о компьютере, включая конфигурацию установленной аппаратуры и программного обеспечения
Hkey_Current_Config	Информация о текущем оборудовании
Hkey_Dyn_Data	Динамические данные о состоянии, используемые процедурами plug-and-play
Hkey_Classes_Root	Информация об OLE, Drag&Drop, клавишах быстрого доступа и пользовательском интерфейсе Windows 95

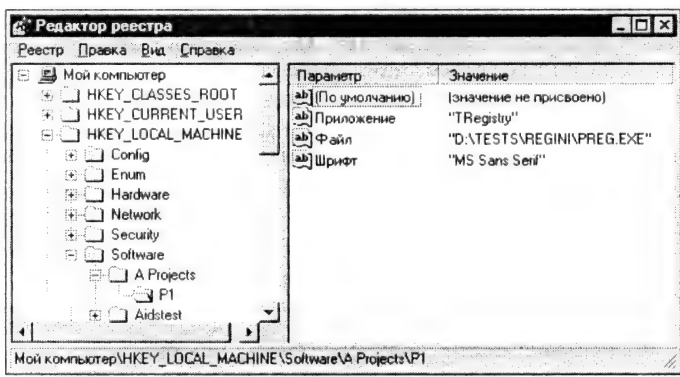
Два ключа верхнего уровня определяют характеристики пользователя:

Hkey_Users	Информация о пользователях, включая установки экрана и приложений
Hkey_Current_User	Информация о пользователе, зарегистрированном в данный момент

Реестр хранится в файле **SYSTEM.DAT** в каталоге Windows. Просмотр и редактирование реестра из Windows осуществляется редактором реестра **Regedit.exe** (рис. 4.25). Только прежде, чем вы будете вручную что-то редактировать в реестре или опробовать изложенные ниже приемы работы с реестром из приложений C++Builder, прислушайтесь к следующему совету.

Совет
Прежде, чем изменять что-то в реестре, сохраните копию его файла **SYSTEM.DAT**, расположенного в каталоге Windows, в каком-то другом каталоге. Это позволит вам в случае неудачного вмешательства в реестр восстановить прежнее состояние этого файла. В противном случае ваши эксперименты могут закончиться плачевно вплоть до необходимости переустанавливать Windows.

Рис. 4.25
Окно редактора реестра
Regedit.exe



Для работы с реестром в C++Builder имеется класс **TRegistry**, описанный в модуле **registry**. Если вы создаете в приложении объект класса **TRegistry** для работы с реестром, не забудьте обеспечить связь с этим модулем директивой

```
#include "registry.hpp";
```

Все ключи в объекте класса **TRegistry** создаются как субключи определенного корневого ключа, записанного в свойстве **RootKey**. По умолчанию **RootKey** = **HKEY_CURRENT_USER**. В каждый момент объект типа **TRegistry** имеет доступ только к одному текущему ключу в иерархии, начинающейся с ключа **RootKey**. Текущий ключ определяется свойством только для чтения **CurrentKey**. Но это значение вам ничего не скажет — это просто некоторое целое значение. А вот свойство **CurrentPath** (тоже только для чтения) содержит строку, включающую имя текущего ключа и путь к нему по дереву. Изменить текущий ключ можно методом **OpenKey**:

```
bool __fastcall OpenKey(const AnsiString Key, bool CanCreate);
```

Этот метод открывает ключ **Key**, делая его текущим для объекта. Параметр **Key** — строка полного пути по дереву ключей к открываемому ключу. Если **Key** — пустая строка, то текущим делается корневой ключ, указанный свойством **RootKey**. Параметр **CanCreate** указывает, должен ли создаваться ключ **Key**, если его

нет в реестре. Ключ открывается или создается с доступом **KEY_ALL_ACCESS**. Создаваемый ключ сохраняется в реестре при последующих запусках системы.

Запись значений параметров в ключ осуществляется группой методов: **WriteInteger**, **WriteFloat**, **WriteBool**, **WriteString** и др. Объявления всех этих методов очень похожи. Например:

```
void __fastcall WriteInteger(const AnsiString Name,
                           int Value);
void __fastcall WriteString(const AnsiString Name,
                           const AnsiString Value);
```

Все они заносят значение **Value** в параметр с именем **Name**. Имеются аналогичные методы чтения: **ReadInteger**, **ReadFloat**, **ReadBool**, **ReadString** и др. Например:

```
int __fastcall ReadInteger(const AnsiString Name);
AnsiString __fastcall ReadString(const AnsiString Name);
```

Посмотрим на примере, как все это можно использовать для установки программы, запоминания ее настроек и для удаления программы.

Сделайте простое тестовое приложение. Перенесите на форму три кнопки **Button** и диалог **FontDialog**. Первая кнопка (назовите ее **BInst** и задайте надпись **Install**) будет имитировать установку программы. Точнее, не саму установку, поскольку копировать файлы с установочной дискеты мы не будем, а только регистрацию нашего приложения в реестре. Вторая кнопка (назовите ее **BUnInst** и задайте надпись **Uninstall**) будет имитировать удаление программы. Тут мы не будем удалять саму программу с диска (жалко ее!), а только удалим из реестра ссылку на нее. А третья кнопка (назовите ее **BFont** и задайте надпись **Font**) будет позволять изменять имя шрифта, используемого в форме, и обеспечит запоминание этого шрифта в реестре, чтобы в дальнейшем при запуске приложения можно было читать эту настройку и задавать ее форме.

Ниже приведен текст этого тестового приложения.

```
#include "registry.hpp";
TRegistry *Reg = new TRegistry;

//-----
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
    //Удаление из памяти объекта Reg
    delete Reg;
}
//-----
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    //Задается корневой каталог объекта Reg
    Reg->RootKey = HKEY_LOCAL_MACHINE;
    if(Reg->KeyExists("\\Software\\A Projects\\P1"))
    {
        /*Если программа была установлена, то читается настройка шрифта*/
        Reg->OpenKey("\\Software\\A Projects\\P1", true);
        Font->Name = Reg->ReadString("Шрифт");
    }
}
//-----

void __fastcall TForm1::BInstClick(TObject *Sender)
{
    //Имитация установки программы
    Reg->OpenKey("\\Software\\A Projects", true);
    Reg->WriteString("Тема", "Мои приложения");
    Reg->OpenKey("\\Software\\A Projects\\P1", true);
```

```

    Reg->WriteString("Приложение", "TRegistry");
    /*Запись в параметр Файл имени и пути к выполняемому файлу*/
    Reg->WriteString("Файл", ParamStr(0));
    //Запись в параметр Шрифт имени шрифта
    Reg->WriteString("Шрифт", Form1->Font->Name);
}
//-----

void __fastcall TForm1::BUnInstClick(TObject *Sender)
{
    //Удаление субключа P1 из регистра
    Reg->DeleteKey("\\Software\\A Projects\\P1");
}

//-----
void __fastcall TForm1::BFontClick(TObject *Sender)
{
    FontDialog1->Font->Assign(Font);
    //Запоминание в реестре имени шрифта
    if (FontDialog1->Execute())
    {
        Font->Assign(FontDialog1->Font);
        if (Reg->OpenKey("\\Software\\A Projects\\P1", false))
        /* Запись имени шрифта только если имеется раздел P1 */
            Reg->WriteString("Шрифт", Form1->Font->Name);
    }
}

```

В начале текста следует подключение к приложению модуля **registry** и создание объекта **Reg** типа **TRegistry**. Этот объект удаляется при закрывании формы в функции **TForm1::FormDestroy**.

При создании формы приложения в процедуре **TForm1::FormCreate** корневым узлом объекта **Reg** задается ключ **HKEY_LOCAL_MACHINE**. Затем с помощью метода **KeyExists** проверяется, существует ли в реестре субключ **\\Software\\A Projects\\P1**. Этот ключ, как мы позже увидим, создается при регистрации нашего приложения в реестре. Так что в настоящем приложении, если бы метод **KeyExists** вернул **false**, надо было бы выдать какое-то предупреждение о том, что приложение не зарегистрировано, и завершить работу.

Обратите внимание, что в текстовых строках символ «\» должен удваиваться: «\\Software\\A Projects\\P1».

Если метод **KeyExists** вернул **true**, то далее в процедуре открывается методом **OpenKey** субключ **\\Software\\A Projects\\P1** и читается методом **ReadString** имя шрифта, занесенное в параметр **Шрифт**, в имя шрифта формы.

Теперь рассмотрим процедуру **TForm1::BInstClick**, имитирующую установку программы. В этой процедуре методами **OpenKey** создается иерархия ключей: **A Projects** и **P1** в субключе **Software** (этот субключ всегда существует в реестре). В ключ **A Projects** заносится один параметр — Тема со значением Мои приложения. Предполагается, что этот субключ будет содержать ссылки на все ваши приложения, зарегистрированные в реестре. В ключ **P1** (субключ вашего регистрируемого проекта) заносится три параметра. Первый параметр — Приложение со значением **TRegistry**. Второй параметр — Файл со значением, равным полному имени файла приложения вместе с путем. Это имя получается из нулевого параметра командной строки, передаваемого функцией **ParamStr(0)**. В третий параметр — Шрифт заносится имя текущего шрифта, используемого в форме. В настоящей программе установки в эту процедуру надо было бы добавить копирование соответствующих файлов с установочной дискеты.

Процедура **TForm1::BUnInstClick** имитирует удаление приложения и ссылки на него (субключ **P1**) из реестра. В настоящем приложении прежде, чем удалять

ключ из реестра, надо было бы прочитать значение параметра **Файл** и удалить этот файл с диска.

Процедура **TForm1::BFontClick** вызывает стандартный диалог выбора шрифта, и если пользователь выбрал шрифт, то он присваивается форме и его имя заносится в реестр. При этом в методе **OpenKey** второй параметр задан равным **false**. Это значит, что если ваше приложение не зарегистрировано (субключ **P1** нет в реестре), то запись шрифта не производится.

Сохраните свое тестовое приложение и запустите его на выполнение. Нажмите кнопку **Install**. После этого запустите программу **Regedit.exe** и посмотрите реестр. Вы увидите там свои ключи и параметры (именно они показаны на рис. 4.25). Щелкните на кнопке **UnInstall**. Вернитесь в окно **Regedit.exe** и выберите в нем команду **Вид | Обновить**. Вы увидите, что ключ **P1** вместе со всеми своими параметрами исчез из дерева. Опять нажмите кнопку **Install**, а затем нажмите кнопку **Font** и выберите шрифт с каким-нибудь другим именем. В окне **Regedit.exe** вы можете увидеть, что значение параметра **Шрифт** изменится. Закройте свое приложение и запустите его повторно. Вы увидите, что на форме применен тот шрифт, который вы регистрировали в реестре. Таким образом, приложение проимитировало установку программы, снятие программы с регистрации и запоминание текущих настроек в реестре.

Когда вы кончите экспериментировать с этим приложением, удалите с помощью **Regedit.exe** из реестра ваш ключ **A Projects**, поскольку приложение удаляло только его субключ **P1**.

4.7.2.2 Работа с файлами .INI

В разделе 4.7.2 рассказывалось, как регистрировать приложение в системном реестре и фиксировать там текущие настройки приложения. Однако, подобная работа с реестром возможна только в 32-разрядных Windows 95/98 и NT. Если же вы хотите, чтобы ваше приложение можно было использовать и в Windows 3.x, то вам надо регистрировать приложение и фиксировать его настройки в файлах типа **.ini**. Для 32-разрядных приложений Microsoft не рекомендует работать с файлами **.ini**. Впрочем, несмотря на это и 32-разрядные приложения, наряду с реестром, часто используют эти файлы. Да и разработки Microsoft не обходятся без этих файлов.

Файлы **.ini** — это текстовые файлы, предназначенные для хранения информации о настройках различных приложений. Информация в файле логически группируется в разделы, каждый из которых начинается оператором заголовка, заключенным в квадратные скобки. Например, **[Desktop]**. В строках, следующих за заголовком содержится информация, относящаяся к данному разделу, в форме:

<ключ>=<значение>

В качестве примера ниже приводится фрагмент файла **ODBC.INI**:

```
[ODBC 32 bit Data Sources]
dBASE Files=Microsoft dBase Driver (*.dbf) (32 bit)
Excel Files=Microsoft Excel Driver (*.xls) (32 bit)
FoxPro Files=Microsoft FoxPro Driver (*.dbf) (32 bit)
Text Files=Microsoft Text Driver (*.txt; *.csv) (32 bit)

[dBASE Files]
Driver32=C:\WINDOWS\SYSTEM\odbcjt32.dll
```

Файлы **.ini**, как правило, хранятся в каталоге **Windows**, который можно найти с помощью функции **GetWindowsDirectory**.

В C++Builder работу с файлами **.ini** проще всего осуществлять с помощью создания в приложении объекта типа **TIniFile**. Этот тип описан в модуле **inifiles**, который надо подключать к приложению оператором **uses** (автоматически это не делается).

При создании объекта типа **TIniFile** в него передается имя файла **.ini**, с которым он связывается. Файл должен существовать до создания объекта.

Для записи значений ключей существует много методов: **WriteString**, **WriteInteger**, **WriteFloat**, **WriteBool** и др. Каждый из них записывает значение соответствующего типа. Объявления всех этих методов очень похожи. Например:

```
void __fastcall WriteString(const AnsiString Section,
                           const AnsiString Ident,
                           const AnsiString Value);
void __fastcall WriteInteger(const AnsiString Section,
                             const AnsiString Ident,
                             int Value);
```

Во всех объявлениях **Section** — раздел файла, **Ident** — ключ этого раздела, **Value** — значение ключа. Если соответствующий раздел или ключ отсутствует в файле, он автоматически создается.

Имеются аналогичные методы чтения: **ReadString**, **ReadInteger**, **ReadFloat**, **ReadBool** и др. Например:

```
AnsiString __fastcall ReadString(const AnsiString Section,
                                const AnsiString Ident,
                                const AnsiString Default);
int __fastcall ReadInteger(const AnsiString Section,
                           const AnsiString Ident,
                           int Default);
```

Методы возвращают значение ключа **Ident** раздела **Section**. Параметр **Default** определяет значение, возвращаемое в случае, если в файле не указано значение соответствующего ключа.

Проверить наличие значения ключа можно методом **ValueExists**, в который передаются имена раздела и ключа. Метод **DeleteKey** удаляет из файла значение указанного ключа в указанном разделе. Проверить наличие в файле необходимого раздела можно методом **SectionExists**. Метод **EraseSection** удаляет из файла указанный раздел вместе со всеми его ключами. Имеется еще ряд методов, которые вы можете посмотреть во встроенной справке C++Builder.

Посмотрим на примере, как все это можно использовать для установки программы, запоминания ее настроек и для удаления программы.

Сделайте простое тестовое приложение, аналогичное рассмотренному в разделе 4.7.2.1. Перенесите на форму три кнопки **Button** и диалог **FontDialog**. Первая кнопка (назовите ее **BInst** и задайте надпись **Install**) будет имитировать установку программы. Точнее, не саму установку, поскольку копировать файлы с установочной дискеты мы не будем, а только создание файла **.ini** в каталоге **Windows**. Вторая кнопка (назовите ее **BUnInst** и задайте надпись **UnInstall**) будет имитировать удаление программы. Тут мы не будем удалять саму программу с диска, а только удалим из каталога **Windows** наш файл **.ini**. А третья кнопка (назовите ее **BFont** и задайте надпись **Font**) будет позволять изменять имя шрифта, используемого в форме, и обеспечит запоминание этого шрифта в файле **.ini**, чтобы в дальнейшем при запуске приложения можно было читать эту настройку и задавать ее форме.

Ниже приведен текст этого тестового приложения.

```
#include "inifiles.hpp";
#include <stdio.h>
TIniFile *Ini;
String sFile;

//-----
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    char APchar[255];
    //Формирование имени файла в каталоге Windows
```

```

GetWindowsDirectory(APchar,255);
sFile = (String)APchar+"\\My.ini";
if(FileExists(sFile))
{
    // Создание объекта Ini
    Ini = new TIniFile(sFile);
    //Чтение в имя шрифта формы значения ключа Шрифт
    Font->Name = Ini->ReadString("Параметры", "Шрифт",
                                "MS Sans Serif");
}
}
//-----
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
    if (Ini == NULL) return;
    //Очистка буфера и запись файла на диск
    Ini->UpdateFile();
    //Освобождение памяти
    delete Ini;
}
//-----
void __fastcall TForm1::BInstClick(TObject *Sender)
{
    FILE *F;
    //Проверка существования файла .ini
    if (! FileExists(sFile))
    {
        //Создание файла .ini
        if ((F = fopen(sFile.c_str(), "w+")) == NULL)
        {
            ShowMessage("Файл не удастся открыть");
            return;
        }
        fclose(F); // закрытие файла
    }
    // Создание объекта Ini
    Ini = new TIniFile(sFile);
    /* Создание раздела Files, ключа main и запись в него
    имени выполняемого файла вместе с путем */
    Ini->WriteString("Files", "main", ParamStr(0));
    /* Создание раздела Параметры, ключа Шрифт и запись в него
    имени шрифта формы */
    Ini->WriteString("Параметры", "Шрифт", Font->Name);
}
//-----
void __fastcall TForm1::BUnInstClick(TObject *Sender)
{
    FILE *F;
    //Удаление с диска файла .ini, если он существует
    if(FileExists(sFile))
        DeleteFile(sFile);
}
//-----
void __fastcall TForm1::BFontClick(TObject *Sender)
{
    //Выбор шрифта
    FontDialog1->Font->Assign(Font);
    if (FontDialog1->Execute())
    {
        //Присваивание выбранного шрифта форме
        Font->Assign(FontDialog1->Font);
        if((Ini != NULL) && Ini->ValueExists("Параметры", "Шрифт"))

```



```
//Запись шрифта в ключ "Шрифт" разделе "Параметры"
Ini->WriteString("Параметры", "Шрифт", Font->Name);
}
}
```

В начале текста следует подключение к приложению модуля **inifiles.hpp**, в котором объявлен тип **TIniFile**, и модуля **stdio.h**, необходимого для операций с файлами. Объявляется объект **Ini** типа **TIniFile** и переменная **sFile**, в которой будет формироваться имя файла и путь к нему. При создании формы приложения в процедуре **TForm1->FormCreate** формируется имя файла («My.ini») вместе с путем к нему — каталогом Windows. Этот путь определяется функцией **GetWindowsDirectory**. Далее функцией **FileExists** проверяется, существует ли этот файл, т.е. проведена ли уже установка программы. Если существует, то создается объект **Ini**, связанный с этим файлом, и значение ключа Шрифт раздела Параметры читается в имя шрифта формы. Тем самым читается настройка, произведенная при предыдущем выполнении приложения.

Теперь рассмотрим процедуру **TForm1->BInstClick**, имитирующую установку программы. В этой процедуре сначала функцией **FileExists** проверяется, существует ли в каталоге Windows файл «My.ini». Если не существует, этот файл (пока пустой) создается функцией **fopen** с параметром «w+». Затем создается связанный с этим файлом объект **Ini**. Последующие операторы записывают в этот файл два раздела Files и Параметры с соответствующими ключами. В ключ **main** записывается имя приложения с путем к нему. Для этого используется функция **ParamStr(0)** (см. раздел 15.7.4 главы 15). В результате в созданный файл записывается, например, такой текст:

```
[Files]
main=D:\TESTS\REGINI\PINI.EXE
```

```
[Параметры]
Шрифт=MS Sans Serif
```

Процедура **TForm1->BUnInstClick** имитирует удаление приложения и его файла настройки. В данном случае просто удаляется файл **.ini**, но в настоящем приложении надо было бы прочитать имя файла (или файлов) приложения из раздела Files и удалить их с диска.

Процедура **TForm1->FormDestroy**, срабатывающая при закрывании формы приложения, сначала методом **UpdateFile** переписывает содержимое объекта **Ini** в файл, а затем методом **Free** удаляет из памяти этот временный объект.

Процедура **TForm1->BFontClick** вызывает стандартный диалог выбора шрифта, и если пользователь выбрал шрифт, то он присваивается форме и его имя заносится в файл **.ini**.

Сохраните свое тестовое приложение и запустите его на выполнение. Нажмите кнопку **Install**. После этого убедитесь в наличии файла «My.ini» в каталоге Windows. Можете воспользоваться для этого программой Windows «Проводник» или любой другой. В частности, можно открыть этот файл просто из среды C++Builder. При нажатии кнопки **UnInstall** файл должен удаляться с диска. Проверьте запись в файл настройки шрифта и чтение ее при последующих запусках. Для этого опять нажмите кнопку **Install**, а затем нажмите кнопку **Font** и выберите шрифт с каким-нибудь другим именем. Потом закройте свое приложение и запустите его повторно. Вы увидите, что на форме применен тот шрифт, который вы зарегистрировали в файле настройки. Таким образом, приложение проимитировало установку программы, удаление программы и запоминание ее текущих настроек.

Глава 5

Графика и мультимедиа

5.1 Построение графических изображений

5.1.1 Использование готовых графических файлов

5.1.1.1 Компонент Image и некоторые его свойства

Нередко возникает потребность украсить свое приложение какими-то картинками. Это может быть графическая заставка, являющаяся логотипом вашего приложения. Или это могут быть фотографии при разработке приложения, работающего с базой данных сотрудников некоего учреждения. В первом случае вам потребуется компонент **Image**, расположенный на странице Additional библиотеки компонентов, во втором — его аналог **DBImage**, связанный с данными и расположенный на странице Data Controls.

Начнем знакомство с этими компонентами. Откройте новое приложение и перенесите на форму компонент **Image**. Его свойство, которое может содержать картинку — **Picture**. Нажмите на кнопку с многоточием около этого свойства или просто сделайте двойной щелчок на **Image**, и перед вами откроется окно Picture Editor (рис. 5.1), позволяющее загрузить в свойство **Picture** какой-нибудь графический файл (кнопка Load), а также сохранить открытый файл под новым именем или в новом каталоге. Щелкните на Load, чтобы загрузить графический файл. Перед вами откроется окно Load Picture, представленное на рис. 5.2. По мере перемещения курсора в списке по графическим файлам в правом окне отображаются содержащиеся в них изображения. Вы можете найти графические файлы в каталоге Images. Он обычно расположен в каталоге ...\\program files\\Common Files\\Borland Shared.

На рис. 5.1 и 5.2 изображена загрузка файла ...\\Images\\Splash\\16Color\\earth.bmp. В окне загрузки графического файла (рис. 5.2) вы можете не только просмотреть изображение, хранящееся в выбираемом файле, но и увидеть размер изображения — цифры в скобках справа сверху. В некоторых случаях, как вы увидите позднее, это важно.

После загрузки файла щелкните на ОК и в вашем компоненте **Image** отобразится выбранная вами картинка. Можете запустить ваше приложение и полюбоваться ею. Впрочем, вы и так увидите картинку, даже не выполняя приложение.

Рис. 5.1

Окно Picture Editor

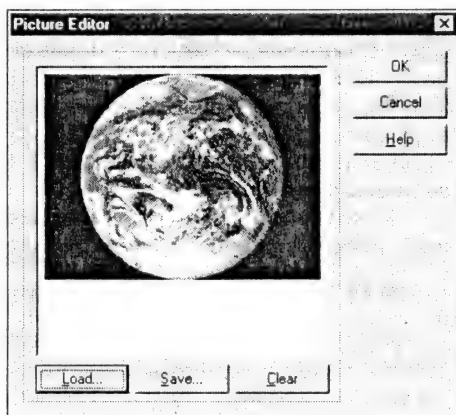
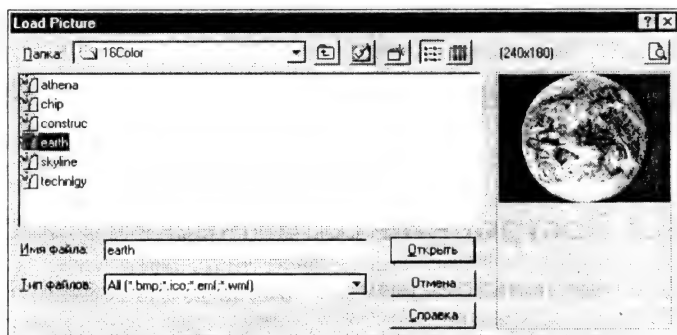


Рис. 5.2

Окно загрузки графического файла



Когда вы в процессе проектирования загрузили картинку из файла в компонент **Image**, он не просто отображает ее, но и сохраняет в приложении. Это дает вам возможность поставлять ваше приложение без отдельного графического файла. Впрочем, как мы увидим позднее, в **Image** можно загружать и внешние графические файлы в процессе выполнения приложения.

Вернемся к рассмотрению свойств компонента **Image**.

Если установить свойство **AutoSize** в **true**, то размер компонента **Image** будет автоматически подгоняться под размер помещенной в него картинки. Если же свойство **AutoSize** установлено в **false**, то изображение может не поместиться в компонент или, наоборот, площадь компонента может оказаться много больше площади изображения.

Другое свойство — **Stretch** позволяет подгонять не компонент под размер рисунка, а рисунок под размер компонента. Установите **AutoSize** в **false**, растяните или сожмите размер компонента **Image** и установите **Stretch** в **true**. Вы увидите, что рисунок займет всю площадь компонента, но поскольку вряд ли реально установить размеры **Image** точно пропорциональными размеру рисунка, то изображение исказится. Устанавливать **Stretch** в **true** может иметь смысл только для каких-то узоров, но не для картинок. Свойство **Stretch** не действует на изображения пиктограмм, которые не могут изменять своих размеров (см. раздел 5.1.1.3).

Свойство — **Center**, установленное в **true**, центрирует изображение на площади **Image**, если размер компонента больше размера рисунка.

Рассмотрим еще одно свойство — **Transparent** (прозрачность). Если **Transparent** равно **true**, то изображение в **Image** становится прозрачным. Это можно использовать для наложения изображений друг на друга. Поместите на форму второй компонент **Image** и загрузите в него другую картинку. Только постарайтесь взять какую-нибудь мало заполненную, контурную картинку. Можете, например, взять картинку из числа помещаемых обычно на кнопки, например, стрелку (файл ...\\program files\\common files\\borland shared\\images\\buttons\\arrow\\r.bmp). Передвиньте ваши **Image** так, чтобы они перекрывали друг друга, и в верхнем компоненте установите **Transparent** равным **true**. Вы увидите, что верхняя картинка перестала заслонять нижнюю. Одно из возможных применений этого свойства — наложение на картинку надписей, выполненных в виде битовой матрицы. Эти надписи можно сделать с помощью встроенной в C++Builder программы Image Editor, которая будет рассмотрена позднее.

Учтите, что свойство **Transparent** действует только на битовые матрицы (о типах графических файлов см. в разделе 5.1.1.3).

5.1.1.2 Простое приложение для просмотра графических файлов

Вы создали приложение, в котором на форме отображается выбранная вами в процессе проектирования картинка. Вы можете легко превратить его в более интересное приложение, в котором пользователь сможет просматривать и загружать

любые графические файлы. Для этого достаточно перенести на форму компонент **OpenPictureDialog**, расположенный в библиотеке на странице Dialogs и вызывающий диалоговое окно открытия и предварительного просмотра изображения (рис. 5.2), а также кнопку, запускающую просмотр или меню с единственным разделом **Файл**.

А теперь вам осталось написать всего один оператор в обработчике щелчка на кнопке или на разделе меню:

```
if (OpenPictureDialog1->Execute())
    Image1->Picture->LoadFromFile (OpenPictureDialog1->FileName);
```

Этот оператор загружает в свойство **Picture** компонента **Image1** файл, выбранный в диалоге пользователем. Выполните свое приложение и проверьте его в работе. Щелкая на кнопке вы можете выбрать любой графический файл и загрузить его в компонент **Image1**.

В таком приложении есть один недостаток — изображения могут быть разных размеров и их положение на форме или будет несимметричным, или они не будут помещаться в окне. Это легко изменить, заставив форму автоматически настраиваться на размеры изображения. Для этого надо установить в компоненте **Image1** свойство **AutoSize** равным **true**, а приведенный ранее оператор изменить следующим образом:

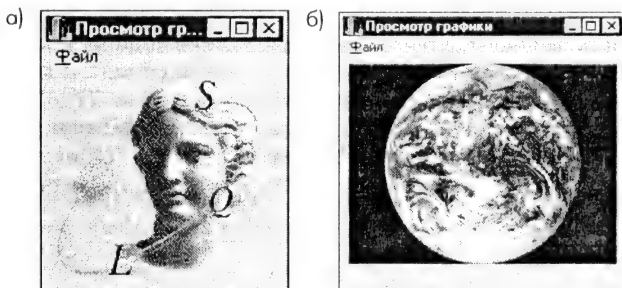
```
if (OpenPictureDialog1->Execute())
{
    Image1->Picture->LoadFromFile (OpenPictureDialog1->FileName);
    Form1->ClientHeight = Image1->Height + 10;
    Image1->Top = Form1->ClientRect.Top
        + (Form1->ClientHeight - Image1->Height) / 2;
    Form1->ClientWidth = Image1->Width + 10;
    Image1->Left = Form1->ClientRect.Left
        + (Form1->ClientWidth - Image1->Width) / 2;
}
```

В этом коде размеры клиентской области формы устанавливаются несколько больше размеров компонента **Image1**, которые в свою очередь адаптируются к размеру картинки благодаря свойству **AutoSize**.

Запустите теперь ваше приложение, и вы увидите (рис. 5.3), что при различных размерах изображения ваше приложение выглядит отлично.

Рис. 5.3

Адаптация формы к размерам изображения



5.1.1.3 Форматы графических файлов

Прежде, чем продвигаться дальше, поговорим немного о форматах графических файлов. С++Builder поддерживает три типа файлов — битовые матрицы, пиктограммы и метафайлы. Все три типа файлов хранят изображения; различные заключаются лишь в способе их хранения внутри файлов и в средствах доступа к ним. Битовая матрица (файл с расширением **.bmp**) отображает цвет каждого пикселя в изображении. При этом информация хранится таким образом, что любой

компьютер может отобразить картинку с разрешающей способностью и количеством цветов, соответствующими его конфигурации.

Пиктограммы (файлы с расширением **.ico**) — это маленькие битовые матрицы. Они повсеместно используются для обозначения значков приложений, в быстрых кнопках, в пунктах меню, в различных списках. Способ хранения изображений в пиктограммах схож с хранением информации в битовых матрицах, но имеются и различия. В частности, пиктограмму невозможно масштабировать, она сохраняет тот размер, в котором была создана.

Метафайлы (Metafiles) хранят не последовательность битов, из которых состоит изображение, а информацию о способе создания картинку. Они хранят последовательности команд рисования, которые и могут быть повторены при воссоздании изображения. Это делает такие файлы, как правило, более компактными, чем битовые матрицы.

C++Builder 5 может работать со следующими файлами:

Тип файла	Расширение
JPEG Image File	.jpg, .jpeg
Битовые матрицы (Bitmaps)	.bmp
Пиктограммы	.ico
Enhanced Metafiles	.emf
Metafiles	.wmf

5.1.1.4 Классы для хранения графических объектов TPicture, TBitmap, TIcon и TMetafile

Выше были рассмотрены типы графических файлов. Для хранения графических объектов, содержащихся в битовых матрицах, пиктограммах и метафайлах, в C++Builder определены соответствующие классы — **TBitmap**, **TIcon** и **TMetafile**. Все они являются производными от абстрактного базового класса графических объектов **TGraphic**. Кроме того определен класс, являющийся надстройкой над **TBitmap**, **TIcon** и **TMetafile** и способный хранить любой из этих объектов. Это класс **TPicture**, с которым вы уже познакомились в начале этой главы. Он имеет свойство **Graphic**, которое может содержать и битовые матрицы, и пиктограммы, и метафайлы. Более того, он может содержать и объекты определенных пользователей графических классов, производных от **TGraphic**. Для доступа к графическому объекту можно использовать свойство **TPicture->Graphic**, но если тип графического объекта известен, то можно непосредственно обращаться к свойствам **TPicture->Bitmap**, **TPicture->Icon** или **TPicture->Metafile**.

Для всех рассмотренных классов определены методы загрузки и сохранения в файл:

```
void __fastcall LoadFromFile(const System::AnsiString Filename);
void __fastcall SaveToFile(const System::AnsiString Filename);
```

При этом для классов **TBitmap**, **TIcon** и **TMetafile** формат файла должен соответствовать классу объекта. Объект класса **TPicture** может оперировать с любым форматом.

Для всех рассмотренных классов определены методы присваивания значений объектам:

```
void __fastcall Assign(TPersistent* Source);
```

Однако, для классов **TBitmap**, **TIcon** и **TMetafile** присваивать можно только значения однородных объектов: соответственно битовых матриц, пиктограмм, метафайлов. При попытке присвоить значения разнородных объектов генерируется ис-

ключение. Класс **TPicture** — универсальный, ему можно присваивать значения объектов любых из остальных трех классов. А значение **TPicture** можно присваивать только тому объекту, тип которого совпадает с типом объекта, хранящегося в нем.

Приведем пример. Часто в приложениях создается объект типа **TBitmap**, значение которого — запомнить содержимое графического изображения и затем восстанавливать его, если оно будет испорчено или изменено пользователем. Код, решающий эту задачу, может иметь вид:

```
// Объявление и создание объекта Bitmap
Graphics::TBitmap *Bitmap = new Graphics::TBitmap();
...

void __fastcall TForm1::FormDestroy(TObject *Sender)
// Уничтожение Bitmap и освобождение памяти
{
    Bitmap->Free();
}
//-----
void __fastcall TForm1::MOpenClick(TObject *Sender)
// Загрузка изображения из файла в Bitmap и Image1
{
    if (OpenPictureDialog1->Execute())
    {
        Bitmap->LoadFromFile(OpenPictureDialog1->FileName);
        Image1->Picture->Assign(Bitmap);
    }
}
//-----
void __fastcall TForm1::MSaveClick(TObject *Sender)
// Сохранение изображения из Image1 в Bitmap
{
    Bitmap->Assign(Image1->Picture);
}
//-----
void __fastcall TForm1::MRestoreClick(TObject *Sender)
// Восстановление изображения в Image1 из Bitmap
{
    Image1->Picture->Assign(Bitmap);
}
```

В этом коде сначала объявляется переменная **Bitmap** типа **TBitmap** и создается соответствующий объект. Если вы создали объект **Bitmap**, то надо не забыть его уничтожить при окончании работы и освободить от него память. Автоматически это не делается. Поэтому надо освобождать память, например, в обработчике события формы **OnDestroy** (процедура **FormDestroy**) методом **Free**:

```
Bitmap->Free();
```

В процедуре **MOpenClick** в объект **Bitmap** методом **LoadFromFile** загружается изображение из выбранного пользователем файла. Затем оператор

```
Image1->Picture->Assign(Bitmap);
```

присваивает значение графического объекта **Bitmap** свойству **Picture** компонента **Image1**. Изображение тем самым делается видимым пользователю.

Этот оператор можно записать иначе:

```
Form1->Image1->Picture->Bitmap->Assign(Bitmap);
```

что даст тот же самый результат.

Если надо переписать в **Bitmap** отредактированное пользователем в **Image1** изображение (о редактировании будет рассказано в последующих разделах), это можно сделать оператором (процедура **MSaveClick**):

```
Bitmap->Assign(Image1->Picture);
```

Если же надо восстановить в **Image1** прежнее изображение, испорченное по каким-то причинам, то это можно сделать оператором (процедура **MRestoreClick**):

```
Image1->Picture->Assign(Bitmap);
```

Таким образом, мы видим, что методом **Assign** можно копировать изображение из одного однотипного графического объекта в другой и обратно.

Загружать изображения можно не только из файлов, но и из ресурсов приложения с помощью методов

```
void __fastcall LoadFromResourceName(int Instance,
                                     const System::AnsiString ResName);
```

или

```
void __fastcall LoadFromResourceID(int Instance, int ResID);
```

где **ResName** — имя графического объекта в файле ресурса, а **ResID** — его идентификатор. Например, оператор

```
Bitmap1->LoadFromResourceName(HInstance, "MYBITMAP");
```

загружает в объект **Bitmap1** из ресурса битовую матрицу с именем «MYBITMAP». Как создавать в ресурсах битовые матрицы и другие графические объекты будет рассказано в разделе 5.1.2.4.

Имеются еще методы загрузки и выгрузки графических объектов в поток и в буфер обмена **Clipboard**, но эти методы используются относительно редко и вы можете посмотреть их в справке по C++Builder.

5.1.2 Редактор Изображений Image Editor

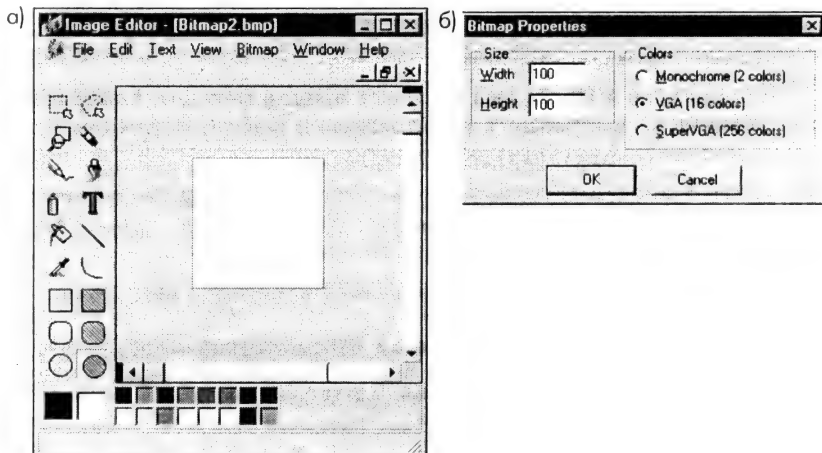
5.1.2.1 Создание файла изображения

В C++Builder имеется встроенный Редактор Изображений — Image Editor, который вызывается командой Tools | Image Editor. Окно Редактора Изображений представлено на рис. 5.4 а. Это сравнительно простой редактор с не очень богатыми возможностями. Он позволяет создавать изображения в виде битовых матриц, пиктограмм, изображений курсоров и не только сохранять созданные изображения в виде файлов, но и сразу включать их в файл ресурсов приложения. В этом и заключается его основное отличие от других, более мощных графических редакторов.

Работа начинается с меню File, в котором вы можете выбрать раздел Open — открыть новый файл изображения или ресурсов, или раздел New — создать новый файл. Если вы выбрали New, то вам предлагается сделать дополнительный выбор, определяющий вид файла, который вы хотите создать:

Рис. 5.4

Окно Редактора Изображений (а) и его вспомогательное окно задания свойств изображения (б)













Resource File (.res)	файл ресурсов
Component Resource File (.dcr)	файл ресурсов компонента
Bitmap File (.bmp)	файл битовой матрицы
Icon File (.ico)	файл пиктограммы
Cursor File (.cur)	файл изображения курсора

Пусть, например, вы хотите создать свой рисунок для битовой матрицы. Тогда, выбрав раздел **Bitmap File**, вы попадаете в окно (рис. 5.4 б), в котором должны выбрать размер (Size) матрицы по горизонтали (Width) и вертикали (Height), а также выбрать набор цветов: 2, 16 или 256. Вероятно, для начала вам будет вполне достаточно 16 цветов.

После сделанного выбора вы увидите в окне Редактора Изображений границы вашего будущего рисунка, как это показано на рис. 5.4 а. Вы можете начинать творить. Раздел меню **View** предоставляет вам возможность увеличить изображение в 2 раза (раздел **Zoom In**), уменьшить ранее увеличенное изображение (раздел **Zoom Out**) или посмотреть изображение в его реальном размере (раздел **Actual Size**).

Расположенная слева инструментальная панель предоставляет вам следующий инструментарий, достаточно типичный для любого графического редактора:

-
-  Выделение прямоугольной области рисунка, которую затем можно передвинуть мышкой, скопировать или вырезать в буфер обмена и т.п.
 -  Выделение произвольной области рисунка, которую затем можно передвинуть мышкой, скопировать или вырезать в буфер обмена и т.п.
 -  Просмотр отдельных пикселей — выделение прямоугольной области рисунка, которая затем увеличивается настолько, что можно работать с отдельными пикселями
 -  Ластик, перемещение которого стирает изображение, окрашивая пиксели вспомогательным цветом, если нажата левая кнопка мыши, или основным цветом, если нажата правая кнопка мыши
 -  Карандаш, перемещение которого наносит линию основным цветом, если нажата левая кнопка мыши, или вспомогательным цветом, если нажата правая кнопка мыши. Толщина линии выбирается из набора, расположенного внизу инструментальной панели
 -  Кисть, перемещение которой окрашивает поверхность основным цветом, если нажата левая кнопка мыши, или вспомогательным цветом, если нажата правая кнопка мыши. Форма кисти выбирается из набора, расположенного внизу инструментальной панели
 -  Пульверизатор. Цвет зависит от нажатой кнопки мыши. Форма пятен выбирается из набора, расположенного внизу инструментальной панели.
 -  Ввод текста. Перед началом ввода или сразу в момент окончания можно пользуясь меню **Text** выбрать тип и размер шрифта
 -  Заполнитель, заливающий выбранным цветом любой нарисованный замкнутый контур или всю поверхность изображения
 -  Индикатор цвета, показывающий цвет пикселя, на который он указывает. Его надо подвести к пикселю, цвет которого вы хотите выбрать, и щелкнуть левой или правой кнопкой мыши, если хотите соответственно назначить этот цвет основным, или вспомогательным
-

Кроме перечисленных инструментов на инструментальной панели вы можете видеть кнопки, соответствующие рисованию прямых линий, дуг, незаполненных и заполненных прямоугольников, прямоугольников со скругленными углами, эллипсов.

При выборе таких инструментов, как карандаш, кисть, пульверизатор, кнопки рисования линий, дуг, незаполненных фигур, внизу появляется палитра, позволяющая выбрать толщину линии или форму кисти.

В нижней части Редактора Изображений (рис. 5.4 а) расположена палитра цветов. В ее левой части имеются два квадрата. Цвет левого из них — назовем его основным, используется, если при рисовании вы нажимаете левую кнопку мыши; цвет правого — вспомогательный, используется, если при рисовании вы нажимаете правую кнопку мыши.

Вот, собственно, и все премудрости. Если вы обладаете художественными способностями (я, увы, ими не обладаю), то можете попробовать нарисовать что-нибудь стоящее. Если же нет, то можете воспользоваться каким-нибудь готовым файлом **.bmp** (команда **File | Open** позволит вам его открыть) и что-то к нему добавить, например, текст. А еще проще — напишите просто текст и сохраните в виде файла **.bmp**. В дальнейшем вы можете наложить его в своем приложении на любой рисунок с помощью свойства **Transparent** компонента **Image**, как было рассказано в разделе 5.1.1.1.

Файл пиктограммы создается аналогично. Вы можете создать, например, несложный файл пиктограммы и затем использовать его как вашу фирменную пиктограмму во всех своих приложениях.

5.1.2.2 Создание пиктограммы для шаблона компонента в библиотеке

Рассмотрим отдельно возможную процедуру создания собственных пиктограмм для библиотеки компонентов, если вы помещаете туда новый компонент или шаблон. Вы научитесь разрабатывать свои шаблоны и компоненты для библиотеки в главе 7. А пока посмотрим, как их можно украсить собственными пиктограммами.

Эти пиктограммы должны иметь формат файла **.bmp** размером 24 на 24. Вы можете, конечно, нарисовать сами нужную пиктограмму, если сумеете. Но поскольку у меня, например, художественные способности отсутствуют, то мне представляется более легким и более качественным следующий путь.

Вы создаете новое приложение **C++Builder**, переносите на форму компонент, похожий на тот, который создаете, и изменяете в нем то, что надо. Например, в разделе 7.2 описана процедура создания шаблона окна редактирования, которое разрешает вводить только цифры. Вы можете взять за основу обычный компонент **Edit**, поместить его на форму и установить в нем текст «0», что укажет пользователю на специфику компонента. Далее надо уменьшить его размеры настолько, чтобы он нормально помещался в размер 24 на 24. При необходимости можно уменьшить соответственно размер шрифта (в этом вам часто поможет помочь шрифт **Small Fonts**). Полезно установить в **false** свойство **AutoSize**. Иначе высота компонента при выполнении приложения будет задана автоматически.

Затем вы запускаете приложение на выполнение и нажимаете клавиши **Alt+Print Screen**. В результате изображение окна вашего приложения будет записано в буфер обмена.

После этого открываете Редактор Изображений (**Tools | Image Editor**), выполняете команду **File | New | Bitmap File (.bmp)** и в открывшемся окне (рис. 5.4 б) задаете размер 24 на 24. Перед вами откроется окно заготовки вашего рисунка. Выполняете команду **Edit | Paste**, которая скопирует изображение из буфера обмена в вашу заготовку рисунка. Остается только мышью сдвинуть его так, чтобы изображение интересующего вас компонента попало в центр рамки рисунка. Затем выполняете команду **File | Save As** и сохраняете рисунок в файле. Далее можете использовать его

как пиктограмму в библиотеке компонентов (см. раздел 7.2, рис. 7.1, в котором загружается только что описанная пиктограмма).

Мы рассмотрели создание пиктограммы для включаемого в библиотеку нового шаблона компонента. Пиктограммы новых компонентов делаются точно так же, но они выполняются не в виде отдельных файлов, а включаются в файлы ресурсов, о которых пойдет речь ниже.

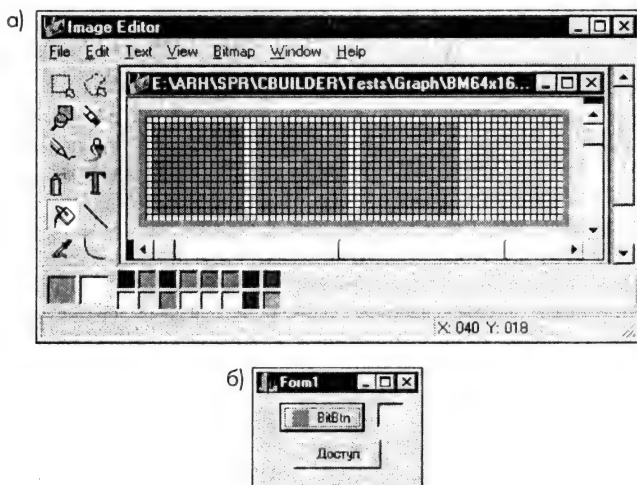
5.1.2.3 Создание пиктограммы для кнопки

Теперь мы попробуем создать пиктограмму для кнопки. Такие кнопки, как **SpeedButton** и **BitBtn**, могут воспринимать пиктограммы, загружаемые в их свойство **Glyph**. Одно изображение пиктограммы имеет размер 16x16. Но в одном файле может содержаться до четырех изображений такого размера. Самое левое соответствует отжатой кнопке. Второе слева соответствует недоступной кнопке, когда ее свойство **Enabled** равно **false**. Третье слева изображение используется при нажатии пользователя на кнопку при ее включении. Четвертое слева изображение используется в кнопках с фиксацией (в **SpeedButton**) для изображения кнопки в нажатом состоянии.

Давайте сделаем пиктограмму по полной программе — на 4 изображения. Создайте новый файл битовой матрицы (File | New | Bitmap File (.bmp)) и в открывшемся окне (рис. 5.4 б) задайте размер 64x16. Разбейте изображение на 4 квадрата, разделенных белыми рамками в один пиксель (рис. 5.5 а), и закрасьте их в последовательности слева направо красным, серым, желтым и зеленым цветами.

Рис. 5.5

Создание пиктограммы для кнопки (а)
и ее использование (б)



При создании файла пиктограмм для кнопок надо иметь в виду, что левый нижний пиксель задает цвет «прозрачности», т.е. цвет, который будет заменяться цветом поверхности кнопки. Поэтому, если вы, например, просто закрасите первый квадрат, не задав ему рамку, то он не будет виден на кнопке.

Сохраните созданный файл и постройте простое приложение, чтобы посмотреть ваши пиктограммы в работе. Перенесите на форму (рис. 5.5 б) три кнопки: **SpeedButton**, **BitBtn** и **Button**. В свойство **Glyph** кнопок **SpeedButton** и **BitBtn** загрузите ваш файл пиктограмм. В свойстве **Caption** кнопки **BitBtn** задайте какую-нибудь надпись, например «BitBtn». При этом вы сможете увидеть, что в свойствах кнопок **NumGlyphs** установится равным 4. Поварьюйте свойствами **Margin** и **Spacing** кнопок, чтобы получить симметричное размещение пиктограмм и надписей. Для кнопки **SpeedButton** установите свойство **GroupIndex** равным 1, а свойство **AllowAllUp** в **true**. В обработчик щелчка кнопки **Button** вставьте код:

```
BitBtn2->Enabled = ! BitBtn2->Enabled;  
SpeedButton2->Enabled = ! SpeedButton2->Enabled;
```

который будет переключать свойство доступности кнопок с пиктограммами. Теперь выполните приложение и посмотрите, как будут меняться цвета кнопок при различных манипуляциях с ними. В отжатом состоянии пиктограммы будут красными. В момент нажатия они окрашиваются в желтый цвет. У нажатой кнопки **SpeedButton** цвет пиктограммы будет зеленым. А в недоступных кнопках цвет пиктограмм будет серым.

5.1.2.4 Работа с файлами ресурсов

В предыдущих разделах рассматривалась последовательность создания новых файлов картинок и пиктограмм. Теперь рассмотрим случай, когда требуется включить картинку, пиктограмму или курсор в файл ресурсов какого-то проекта или компонента. Файлы ресурсов проектов имеют расширение **.res** и содержат битовые матрицы (**.bmp**), пиктограммы (**.ico**), изображения курсоров (**.cur**), используемые в проекте. Файлы ресурсов компонентов имеют расширение **.dcr** (dynamic component resource — динамические ресурсы компонента) и могут включать такие же элементы, как и файлы **.res**.

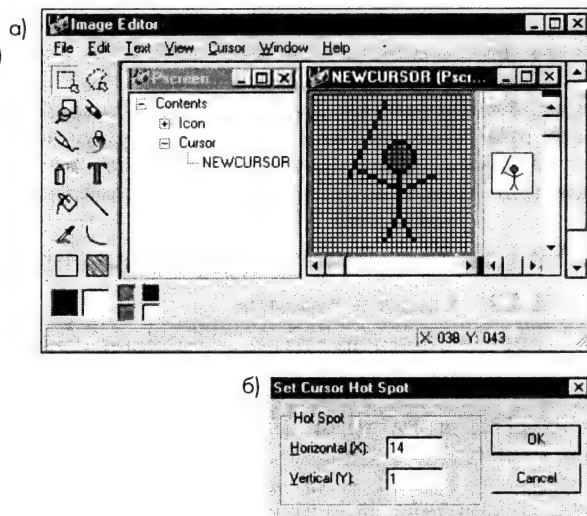
В некоторых случаях включение изображений в файл ресурса — единственная возможность решить ту или иную задачу. Например, если вы хотите ввести в своем приложении какой-то нестандартный курсор, это можно сделать, зарегистрировав его с помощью функции **LoadCursor** в свойстве **Cursors** компонента **Screen** (необходимые для этого операции подробно обсуждаются в разделе 4.5.6). Однако, для использования своего курсора надо сначала создать его и включить в ресурс приложения с помощью Редактора Изображений. Другой пример — создание пиктограммы для нового компонента, включаемого вами в библиотеку C++Builder (см. разделы 5.1.2.2 и 7.3). Эта пиктограмма также должна быть создана не в виде отдельного файла, а как элемент ресурса компонента.

Работа с файлом ресурса приложения в Редакторе Изображений обычно начинается командой **File | Open**, открывающей файл ресурсов приложения **.res**, в котором вы хотите что-то изменить. Перед вами открывается окно, содержащее структуру файла в виде дерева (рис. 5.6 а). Сначала в нем может быть только один узел — **Icon**, содержащий вершину **MAINICON**, соответствующую стандартной пиктограмме приложения. Добавить новые узлы вы можете с помощью команды **Resource | New** (ее вы можете найти и во всплывающем меню Редактора Изображений), выполняя которую вам предоставляется возможность выбрать один из типов элементов: **Bitmap**, **Cursor** или **Icon**. Пусть, например, вы хотите использовать в своем приложении нестандартный курсор в виде человечка с указкой. Вы выполняете **Resource | New | Cursor** и в дереве структуры файла ресурсов приложения появляется новая вершина. Вы должны задать ей то имя, которое в дальнейшем будете использовать в приложении при регистрации курсора функцией **LoadCursor**. Затем вы щелкаете на созданной вершине и вам открывается заготовка изображения курсора, в которой вы рисуете нужную картинку.

При рисовании курсора в главном меню Редактора Изображений появляется раздел **Cursor** (см. рис. 5.6 а) с двумя подразделами: **Set Hot Spot** — указание горячей точки, и **Test** — тестирование. Горячая точка — это та точка изображения курсора, координатами которой являются параметры **X** и **Y**, передающиеся в обработчики событий мыши (см. раздел 4.3.1.2). При выборе раздела меню **Set Hot Spot** открывается диалоговое окно (см. рис. 5.6 б), в котором вы должны задать горизонтальную и вертикальную координаты горячей точки. Координаты задаются в пикселях, т.е. чтобы правильно указать координату надо посчитать, на сколько клеточек требуемая точка отстоит от левого верхнего угла рисунка. Например, для курсора на рис. 5.6 а в качестве горячей точки, очевидно, надо задать точку конца указки.

Рис. 5.6

Структура файла ресурсов приложения (а)
и окно задания горячей точки курсора (б)



После задания горячей точки вы можете выбрать раздел меню **Cursor | Test**. Перед вами откроется окно, в котором курсор приобретет нарисованный вами вид. Нажав кнопку мыши и передвинув курсор, вы увидите, что за курсором тянется нарисованная линия. Вы можете видеть, из правильной ли точки курсора выходит эта линия, и при необходимости можете подкорректировать координаты горячей точки или само изображение курсора.

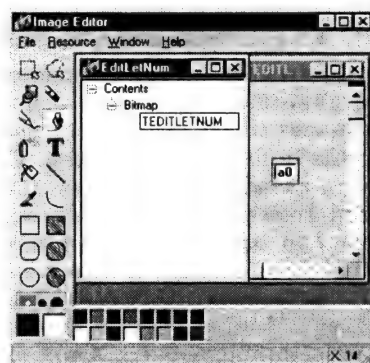
Когда вы завершили создание курсора, надо вернуться в окно структуры файла ресурса и выполнить команду **File | Save**.

Несколько иначе, но подобным же образом создаются файлы ресурсов компонентов. Эта процедура иллюстрируется рисунком 5.7. Начинается работа с команды **File | New | Component Resource File (.dcr)**. Затем, если вы хотите создать пиктограмму для регистрации вашего нового компонента в библиотеке, вы должны выполнить команду **Resource | New | Bitmap** и в открывшемся диалоговом окне задать размер картинки — 24 на 24. Только такой размер воспримется при регистрации вашего компонента в библиотеке.

Затем вы должны назвать соответствующую вершину в дереве структуры файла ресурсов тем же именем, которое имеет вводимый вами класс компонента. В противном случае C++Builder не воспримет это изображение как пиктограмму вашего компонента. И последнее требование. Ваш файл **.dcr** вы должна сохранить с именем, совпадающим с именем файла модуля, в котором вы создаете компонент, и в том же самом каталоге, в котором хранится этот файл модуля. Например, рис. 5.7 подразумевает, что вы создаете компонент, описанный в разделе 7.3.

Рис. 5.7

Структура файла ресурсов компонента



Класс вашего компонента назван **TEditLetNum**, а имя модуля, в котором вы создаете свой компонент, — **EditLetNum**.

Только при соблюдении всех перечисленных выше условий ваше изображение будет воспринято **C++Builder** как пиктограмма компонента и она будет использована при установке компонента на странице библиотеки.

А само создание изображения ничем не отличается от создания пиктограммы шаблона компонента, подробно описанного в разделе 5.1.2.2.

5.1.3 Канва — холст для рисования

5.1.3.1 Канва и пиксели

Многие компоненты в **C++Builder** имеют свойство **Canvas** (канва, холст), представляющее собой область компонента, на которой можно рисовать или отображать готовые изображения. Это свойство имеют формы, графические компоненты **Image**, **PaintBox**, **Bitmap** и многие другие. Канва содержит свойства и методы, существенно упрощающие графику **C++Builder**. Все сложные взаимодействия с системой спрятаны для пользователя, так что рисовать в **C++Builder** может человек, совершенно не искушенный в машинной графике.

Каждая точка канвы имеет координаты **X** и **Y**. Система координат канвы, как и везде в **C++Builder**, имеет началом левый верхний угол канвы. Координата **X** возрастает при перемещении слева направо, а координата **Y** — при перемещении сверху вниз.

С координатами вы уже имели дело многократно, но пока вас не очень интересовало, что стоит за ними, в каких единицах они измеряются. Координаты измеряются в пикселях. *Пиксель* — это наименьший элемент поверхности рисунка, с которым можно манипулировать. Важнейшее свойство пикселя — его цвет. Для описания цвета используется тип **TColor**. С цветом вы встречаетесь практически в каждом компоненте и знаете, что в **C++Builder** определено множество констант типа **TColor**. Одни из них непосредственно определяют цвета (например **clBlue** — синий), другие определяют цвета элементов окон, которые могут меняться в зависимости от выбранной пользователем палитры цветов **Windows** (например, **clBtnFace** — цвет поверхности кнопок). Полный перечень этих констант с пояснениями см. в главе 16.

Но для графики иногда этих предопределенных констант не хватает. Вам могут понадобиться такие оттенки, которых нет в стандартных палитрах. В этом случае можно задавать цвет 4-байтовым шестнадцатеричным числом, три младших разряда которого представляют собой интенсивности синего, зеленого и красного цвета соответственно. Например, значение **\$00FF0000** соответствует чистому синему цвету, **\$0000FF00** — чистому зеленому, **\$000000FF** — чистому красному. **\$00000000** — черный цвет, **\$00FFFFFF** — белый. Более подробно об этом вы можете посмотреть в соответствующем разделе главы 15.

5.1.3.2 Рисование по пикселям

Рисовать на канве можно разными способами. Первый вариант — рисование по пикселям. Для этого используется свойство канвы **Pixels**. Это свойство представляет собой двумерный массив **Canvas->Pixels[int X][int Y]**, который отвечает за цвета канвы. Например, **Canvas->Pixels[10][20]** соответствует цвету пикселя, 10-го слева и 20-го сверху. С массивом пикселей можно обращаться как с любым свойством: изменять цвет, задавая пикселю новое значение, или определять его цвет по хранящемуся в нем значению. Например, **Canvas->Pixels[10][20] = clBlack** — это задание пикселю черного цвета.

Давайте попробуем нарисовать график некоторой функции **F(X)** на канве компонента **Image1**, если известен диапазон ее изменения **Ymax** и **Ymin** и диапазон изменения аргумента **Xmin** и **Xmax**. Это можно сделать такой процедурой:

```

float X,Y;           // координаты функции
int PX,PY;           // координаты пикселей
for (PX = 0; PX <= Image1->Width; PX++)
{
    //X - координата, соответствующая пикселю с координатой PX
    X = Xmin + PX * (Xmax - Xmin) / Image1->Width;
    Y = F(X);
    //PY - координата пикселя, соответствующая координате Y
    PY = Image1->Height - (Y - Ymin) * Image1->Height / (Ymax - Ymin);
    //Устанавливается черный цвет выбранного пикселя
    Image1->Canvas->Pixels[PX][PY] = clBlack;
}

```

В этом коде вводятся переменные **X** и **Y**, являющиеся значениями аргумента и функции, а также переменные **PX** и **PY**, являющиеся координатами пикселей, соответствующими **X** и **Y**. Сама процедура состоит из цикла по всем значениям горизонтальной координаты пикселей **PX** компонента **Image1**. Сначала выбранное значение **PX** пересчитывается в соответствующее значение **X**. Затем производится вызов функции и определяется ее значение **Y**. Это значение пересчитывается в вертикальную координату пикселя **PY**. И в заключение цвет пикселя с координатами (**PX**, **PY**) устанавливается черным.

Попробуйте создать соответствующее приложение и посмотреть, как оно работает. Пусть для простоты мы будем ориентироваться на функцию $\sin(X)$, для которой **Xmin**=0, **Xmax**=4 π (2 периода в радианах), **Ymin**=-1, **Ymax**=1.

Начните новый проект, поместите на него компонент **Image** и кнопку с надписью «Нарисовать», в обработчик события **OnClick** которой запишите код, аналогичный приведенному выше, но конкретизирующий функцию:

```

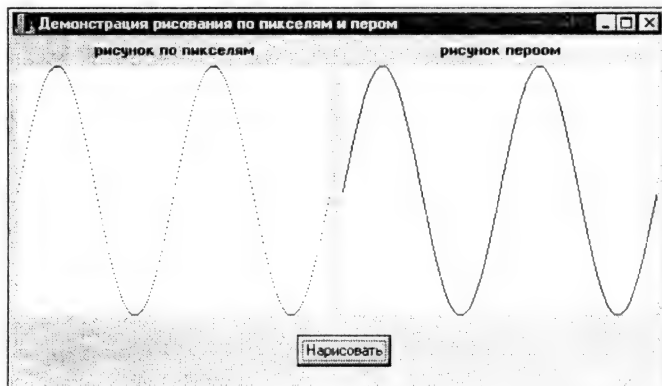
#define Pi 3.14159
float X,Y;           // координаты функции
int PX,PY;           // координаты пикселей
for (PX = 0; PX <= Image1->Width; PX++)
{
    //X - координата, соответствующая пикселю с координатой PX
    X = PX * 4 * Pi / Image1->Width;
    Y = sin(X);
    //PY - координата пикселя, соответствующая координате Y
    PY = Image1->Height - (Y+1) * Image1->Height / 2;
    //Устанавливается черный цвет выбранного пикселя
    Image1->Canvas->Pixels[PX][PY] = clBlack;
}

```

Откомпилируйте ваш проект, сохраните его (мы еще к нему вернемся) и выполните. Результат представлен на рис. 5.8 в его левой части. Правая часть будет рассмотрена в следующем разделе.

Рис. 5.8

Рисование по пикселям и функцией **LineTo**



5.1.3.3 Рисование с помощью пера Pen

У канвы имеется свойство **Pen** — перо. Это объект, в свою очередь имеющий ряд свойств. Одно из них уже известное вам свойство **Color** — цвет, которым наносится рисунок. Второе свойство — **Width** (ширина линии). Ширина задается в пикселях. По умолчанию ширина равна 1.

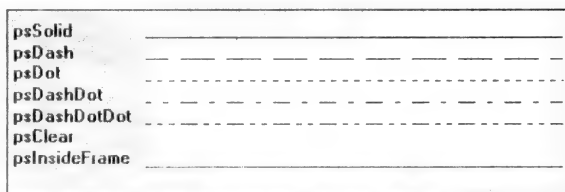
Свойство **Style** определяет вид линии. Это свойство может принимать следующие значения:

psSolid	Сплошная линия
psDash	Штриховая линия
psDot	Пунктирная линия
psDashDot	Штрих-пунктирная линия
psDashDotDot	Линия, чередующая штрих и два пунктира
psClear	Отсутствие линии
psInsideFrame	Сплошная линия, но при Width > 1 допускающая цвета, отличные от палитры Windows

Примеры линий всех стилей приведены на рис. 5.9.

Рис. 5.9

Линии различных стилей



Все стили со штрихами и пунктирами доступны только при **Width = 1**. В противном случае линии этих стилей рисуются как сплошные.

Стиль **psInsideFrame** — единственный, который допускает произвольные цвета. Цвет линии при остальных стилях округляется до ближайшего из палитры Windows.

У канвы имеется свойство **PenPos**. Это свойство определяет в координатах канвы текущую позицию пера. Перемещение пера без прорисовки линии, т.е. изменение **PenPos**, производится методом канвы **MoveTo(X,Y)**. Здесь **(X,Y)** — координаты точки, в которую перемещается перо. Эта текущая точка становится исходной, от которой методом **LineTo(X,Y)** можно провести линию в точку с координатами **(X,Y)**. При этом текущая точка перемещается в конечную точку линии и новый вызов **LineTo** будет проводить точку из этой новой текущей точки.

Давайте попробуем нарисовать пером график синуса из предыдущего примера. Откройте прежний проект, добавьте на него еще один компонент **Image** и разместите компоненты так, как показано выше на рис. 5.8. Размеры обоих компонентов **Image** должны быть абсолютно одинаковы, так как на этом для экономии размера кода и вашего труда основана программа, которую мы напишем. Сделать размеры компонентов абсолютно одинаковыми легко, выделив их оба и воспользовавшись командой всплывающего меню **Size**.

Затем в уже написанном вами обработчике щелчка на кнопке добавьте перед началом цикла оператор

```
Image2->Canvas->MoveTo(0,Image2->Height / 2);
```

который переводит перо в начало координат второго графика — на левый край канвы в середину ее высоты. А в конце цикла добавьте оператор

```
Image2->Canvas->LineTo(PX, PY);
```

который рисует на втором графике линию, соединяющую соседние точки. Иначе говоря, теперь ваш код должен иметь вид:

```
#define Pi 3.14159
float X,Y;           // координаты функции
int PX,PY;           // координаты пикселей
Image2->Canvas->MoveTo(0,Image2->Height / 2);
for (PX = 0; PX <= Image1->Width; PX++)
{
    //X — координата, соответствующая пикселю с координатой PX
    X = PX * 4 * Pi / Image1->Width;
    Y = sin(X);
    //PY — координата пикселя, соответствующая координате Y
    PY = Image1->Height - (Y+1) * Image1->Height / 2;
    //Устанавливается черный цвет выбранного пикселя
    Image1->Canvas->Pixels[PX][PY] = clBlack;
    //Проводится линия на втором графике
    Image2->Canvas->LineTo(PX, PY);
}
```

Для экономии кода мы воспользовались тем, что оба графика у нас абсолютно одинакового размера и, следовательно, пересчет координат достаточно провести для одного из них, а потом воспользоваться этими координатами для рисования обоих графиков.

Откомпилируйте приложение и выполните его. Вы получите результат, представленный на рис. 5.8. Легко видеть, что качество двух одинаковых графиков сильно различается. В левом на крутых участках сплошной линии нет — она распадается на отдельные точки — пиксели. А правый график весь сплошной. Это показывает, что при прочих равных условиях рисовать лучше не по пикселям, а пером.

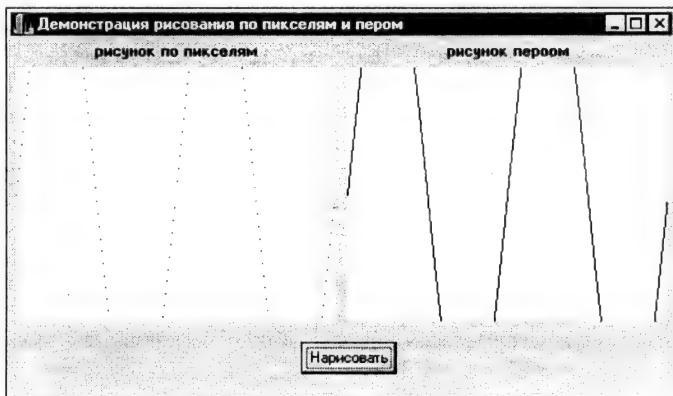
Отметим еще одно ценное свойство компонента **Image** и его канвы. Вы можете задавать координаты пикселей, выходящие за пределы размеров канвы, и ничего страшного при этом не случится. Это позволяет не заботиться о том, какая часть рисунка попадает в рамку **Image**, а какая нет. Вы можете легко проверить это, увеличив, например, вдвое размах вашей синусоиды. Для этого достаточно изменить оператор, задающий значение **Y**, на следующий:

```
Y = 2 * sin(X);
```

Вы получите результат, показанный на рис. 5.10. Изобразилась только та часть рисунка, которая помещается в рамку канвы. Это позволяет легко осуществ-

Рис. 5.10

Изображение при размерах изображения, превышающих размер канвы



лять приложения, в которых пользователю предоставляется возможность увеличивать и просматривать в деталях какие-то фрагменты графиков.

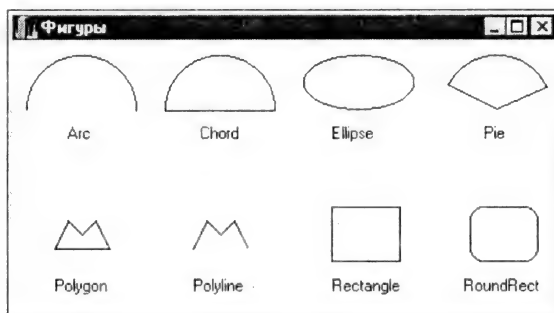
Перо может рисовать не только прямые линии, но и фигуры. Ниже перечислены некоторые из методов канвы, использующие перо для рисования фигур:

Arc	Рисует дугу окружности или эллипса
Chord	Рисует замкнутую фигуру, ограниченную дугой окружности или эллипса и хордой
Ellipse	Рисует окружность или эллипс
Pie	Рисует сектор окружности или эллипса
Polygon	Рисует замкнутую фигуру с кусочно-линейной границей
Polyline	Рисует кусочно-линейную кривую
Rectangle	Рисует прямоугольник
RoundRect	Рисует прямоугольник со скругленными углами

Примеры фигур приведены на рис. 5.11.

Рис. 5.11

Примеры фигур, нарисованных пером



Ниже приведен текст процедуры, которая рисовала фигуры, показанные на рис. 5.11. Этот текст поможет вам понять методы, осуществляющие рисование фигур. Подробное описание методов рисования вы найдете в главе 16.

```
Image1->Canvas->Font->Style < fsBold;
Image1->Canvas->Arc(10,10,90,90,90,50,10,50);
Image1->Canvas->TextOut(40,60,"Arc");
Image1->Canvas->Chord(110,10,190,90,190,50,110,50);
Image1->Canvas->TextOut(135,60,"Chord");
Image1->Canvas->Ellipse(210,10,290,50);
Image1->Canvas->TextOut(230,60,"Ellipse");
Image1->Canvas->Pie(310,10,390,90,390,30,310,30);
Image1->Canvas->TextOut(340,60,"Pie");
TPoint points[5];
points[0] = Point(30,150);
points[1] = Point(40,130);
points[2] = Point(50,140);
points[3] = Point(60,130);
points[4] = Point(70,150);
Image1->Canvas->Polygon(points,4);
Image1->Canvas->TextOut(30,170,"Polygon");
points[0].x += 100;
points[1].x += 100;
points[2].x += 100;
```

```

points[3].x += 100;
points[4].x += 100;
Image1->Canvas->Polyline(points,4);
Image1->Canvas->TextOut(130,170,"Polyline");
Image1->Canvas->Rectangle(230,120,280,160);
Image1->Canvas->TextOut(230,170,"Rectangle");
Image1->Canvas->RoundRect(330,120,380,160,20,20);
Image1->Canvas->TextOut(325,170,"RoundRect");

```

Проглядите этот текст. Подобный пример вы можете увидеть на прилагаемом к книге диске. Там вы можете не только просмотреть текст, но и поменять параметры методов, чтобы лучше понять, как они работают.

Для вывода текста на канву в приведенном примере использован метод **TextOut**, синтаксис которого:

```
void __fastcall TextOut(int X, int Y, const System::AnsiString Text);
```

Имеется еще несколько методов вывода текста, которые применяются реже. Все методы вывода текста используют свойство канвы **Font** — шрифт, с которым вы уже знакомы. В частности, в приведенном примере с помощью этого свойства установлен жирный шрифт надписей.

5.1.3.4 Brush — кисть

У канвы имеется свойство **Brush** — кисть. Это свойство определяет фон и заполнение замкнутых фигур на канве. **Brush** — это объект, имеющий в свою очередь ряд свойств. Свойство **Color** определяет цвет заполнения. Свойство **Style** определяет шаблон заполнения (штриховку). Возможные значения этого свойства см. в соответствующем разделе главы 16.

Имеется еще одно свойство кисти — **Bitmap**, являющееся указателем на объект типа **TBitmap** и определяющее нестандартное заполнение заданным шаблоном. Шаблон задается битовой матрицей размером 8 на 8. Если для кисти задан шаблон **Bitmap**, то заполнение производится именно этим шаблоном, независимо от значения свойства **Style**.

Шаблон **Bitmap** может создаваться в процессе выполнения приложения или, например, загружаться из файла, как в приведенном ниже примере, в котором фон формы заполняется загруженным шаблоном:

```

Graphics::TBitmap *BrushBmp = new Graphics::TBitmap;
try
{
    BrushBmp->LoadFromFile(«MyBitmap.bmp»);
    Form1->Canvas->Brush->Bitmap = BrushBmp;
    Form1->Canvas->FillRect(Rect(0,0,Form1->Width,Form1->Height));
}
__finally
{
    Form1->Canvas->Brush->Bitmap = NULL;
    delete BrushBmp;
}

```

В этом примере создается объект **BrushBmp** типа **TBitmap** и в него загружается битовая матрица из файла с именем **MyBitmap.bmp**. Затем свойству **Form1->Canvas->Brush->Bitmap** присваивается указатель на этот объект. После этого загруженный шаблон можно использовать для заполнения фигур на канве формы. Метод **FillRect** рисует на канве заполненный шаблоном прямоугольник, занимающий всю площадь формы. Если подобный код вставить в обработчик события формы **OnResize**, то и при изменении пользователем размеров формы ее поверхность будет вся заполнена шаблоном. После этого (а также в случае генерации каких-то исключений) **Bitmap** присваивается значение **NULL**, после чего заполнение опять

начинает определяться свойством **Style**. Затем объект **BrushBmp** уничтожается, чтобы освободить занимаемую им память.

В приведенном примере использован метод канвы **FillRect**, объявленный как

```
void __fastcall FillRect(const Windows::TRect &Rect);
```

Он заполняет заданным стилем или шаблоном прямоугольную область, заданную параметром **Rect**. Этот параметр имеет тип **TRect**. Для его задания проще всего использовать функцию **Rect(X1,Y1,X2,Y2)**, возвращающую структуру **Rect** с координатами углов, заданных параметрами (**X1**, **Y1**) и (**X2**, **Y2**).

Функцию **FillRect** удобно, в частности, использовать для стирания изображения. Например, оператор

```
Image1->Canvas->FillRect(Rect(0,0,Image1->Width,Image1->Height));
```

очищает всю площадь канвы компонента **Image1**.

Кисть участвует в заполнении фигур не только с помощью этой функции. Все перечисленные ранее методы рисования замкнутых фигур тоже заполняют их с помощью кисти. Это относится к методам **Chord**, **Ellipse**, **Pie**, **Polygon** и др.

Имеется еще один интересный метод, работающий с кистью. Это метод **FloodFill**, который заполняет замкнутую область на канве. Этот метод определен следующим образом:

```
void __fastcall FloodFill(int X, int Y, TColor Color,
                          TFillStyle FillStyle);
```

Точка с координатами **X** и **Y** является произвольной внутренней точкой заполняемой области, которая может иметь произвольную форму. Граница этой области определяется сочетанием параметров **Color** и **FillStyle**. Параметр **Color** указывает цвет, который используется при определении границы заполняемой области, а параметр **FillStyle** определяет, как именно по этому цвету определяется граница. Параметр **FillStyle** может принимать одно из двух следующих значений: **fsSurface** или **fsBorder**. Если **FillStyle = fsSurface**, то заполняется область, окрашенная цветом **Color**, а на других цветах метод останавливается. Если **FillStyle = fsBorder**, то наоборот, заполняется область окрашенная любыми цветами, не равными **Color**, а на цвете **Color** метод останавливается.

Для определения области закрашивания можно использовать координаты и цвет одного из пикселей, расположенных внутри области (если **FillStyle = fsSurface**) или снаружи ее (если **FillStyle = fsBorder**). В следующем разделе 5.1.4 будет приведен пример использования метода **FloodFill**.

Имеется еще один метод канвы, связанный с кистью. Это метод **FrameRect**. Он рисует на канве текущей кистью прямоугольную рамку, не закрашивая ее. Синтаксис метода **FrameRect**:

```
void __fastcall FrameRect(const Windows::TRect &Rect);
```

Параметр **Rect** определяет позицию и размеры прямоугольной рамки. Толщина рамки — 1 пиксель. Область внутри рамки кистью не заполняется. Метод **FrameRect** отличается от рассмотренного ранее метода **Rectangle** тем, что рамка рисуется цветом кисти (в методе **Rectangle** — цветом пера) и область не закрашивается (в методе **Rectangle** закрашивается).

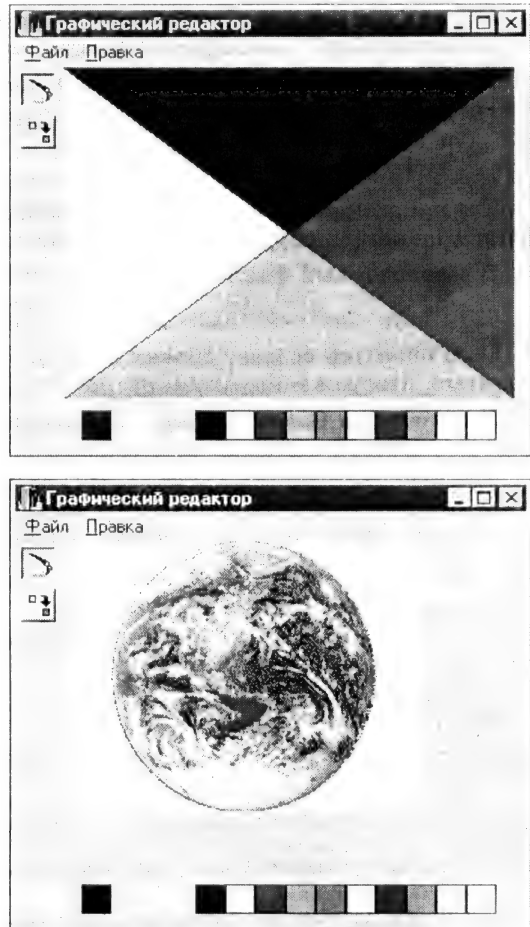
5.1.4 Пример построения собственного простого графического редактора

Давайте попробуем использовать полученные нами сведения для построения собственного простенького графического редактора, воспроизводящего некоторые функции настоящих редакторов.

1. Начните новое приложение.
2. Перенесите на форму два компонента типа **TImage** и расположите их в нижней левой части формы (рис. 5.12), придав квадратную форму, например, размером 20 на 20. Это будут окна основного и вспомогательного цветов. Имена этих компонентов будут **Image1** и **Image2**.

Рис. 5.12

Редактор изображений в работе: тестовая картинка (а) и работа с изображением, загруженным из файла (б)



3. Перенесите на форму еще один компонент типа **TImage** и расположите его в верхней части формы, несколько отступив от левого края, где у нас будет инструментальная панель, и растянув так, чтобы он занимал основную часть формы. Это будет холст для рисования. Имя этого компонента будет **Image3**.
4. Перенесите на форму еще один компонент типа **TImage** и расположите его внизу правее первых двух на одном с ними уровне. Это будет палитра цветов. Ее высоту задайте той же, что у первых двух компонентов, а длину — в 10 раз большую. Имя этого компонента будет **Image4**.
5. Перенесите на форму кнопку типа **TSpeedButton** и расположите ее в верхнем левом углу формы. Эта кнопка будет соответствовать кисти — типичному инструменту графических редакторов. Назовите ее **SBBrush**. Установите у кнопки свойство **GroupIndex** равным 1 и свойство **AllowAllUp** в **true**. Эти свойства обеспечат кнопке возможность фиксироваться в нажатом и не нажатом состоя-

нии (см. раздел 3.5.2). Желательно загрузить в свойство **Glyph** пиктограмму кисти (файл ..\Images\Buttons\brush.bmp).

6. Перенесите на форму еще одну кнопку типа **TSpeedButton** и расположите ее ниже **SBBrush**. Эта кнопка будет соответствовать указателю цвета пикселя рисунка. Назовите ее **SBColor**. Установите у этой кнопки, как и у предыдущей, свойство **GroupIndex** равным 1 (это обеспечит, что только одна из двух кнопок может быть нажата) и свойство **AllowAllUp** в **true**. Желательно загрузить в свойство **Glyph** пиктограмму (например, файл ..\Images\Buttons\one2one.bmp).
7. Перенесите на форму диалог **OpenPictureDialog**.
8. Перенесите на форму главное меню **MainMenu**. В меню задайте раздел Файл с подразделом Открыть. Назовите этот подраздел **MOpen**. Задайте еще один раздел — Правка с подразделом Отменить. Назовите этот подраздел **Undo**.

Теперь размещение компонентов закончено и можно писать обработчики событий. Начнем со второстепенных обработчиков.

9. В заголовочный файл модуля включите оператор:

```
Graphics::TBitmap *BitMap = new Graphics::TBitmap;
```

Этот оператор создает объект **BitMap** типа **TBitmap**. В этом объекте будет храниться изображение, чтобы его можно было восстановить командой Отменить.

10. Для события **OnCreate** формы напишите обработчик вида:

```
// задание свойств кисти основного и вспомогательного цветов
Image1->Canvas->Brush->Color = clBlack;
Image2->Canvas->Brush->Color = clWhite;
// заполнение окон основного и вспомогательного цветов
Image1->Canvas->FillRect(Rect(0,0,Image1->Width,
                             Image1->Height));
Image2->Canvas->FillRect(Rect(0,0,Image2->Width,
                             Image2->Height));
// задание ширины элемента палитры цветов
int HW = Image4->Width / 10;
// закраска элементов палитры цветов
for(int i = 1; i <=10; i++)
{
    switch (i)
    {
        case 1:Image4->Canvas->Brush->Color = clBlack;
                break;
        case 2:Image4->Canvas->Brush->Color = clAqua;
                break;
        case 3:Image4->Canvas->Brush->Color = clBlue;
                break;
        case 4:Image4->Canvas->Brush->Color = clFuchsia;
                break;
        case 5:Image4->Canvas->Brush->Color = clGreen;
                break;
        case 6:Image4->Canvas->Brush->Color = clLime;
                break;
        case 7:Image4->Canvas->Brush->Color = clMaroon;
                break;
        case 8:Image4->Canvas->Brush->Color = clRed;
                break;
        case 9:Image4->Canvas->Brush->Color = clYellow;
                break;
        case 10:Image4->Canvas->Brush->Color = clWhite;
    }
    Image4->Canvas->Rectangle((i-1)*HW,0,i*HW,
                             Image4->Height);
}
```



```

    }
    // рисование креста на холсте — только для тестирования
    Image3->Canvas->MoveTo(0,0);
    Image3->Canvas->LineTo(Image3->Width,Image3->Height);
    Image3->Canvas->MoveTo(0,Image3->Height);
    Image3->Canvas->LineTo(Image3->Width,0);
    BitMap->Assign(Image3->Picture);

```

Обработчик длинный, но смысл его достаточно прост. Сначала задаются свойства кисти окон основного (**Image1**) и вспомогательного (**Image2**) цветов: черный и белый. Окна заливаются соответствующим цветом с помощью функции **FillRect**. После этого формируется палитра цветов: для каждого элемента палитры задается свой цвет и элемент заполняется этим цветом с помощью функции **Rectangle**. Затем на холсте **Image3** рисуется крест (рис. 5.12 а). Он имеет чисто демонстрационный характер, чтобы можно было протестировать программу на простом изображении. В реальной программе этот рисунок не нужен и, если хотите, можно эти операторы не писать. В конце нарисованное на канве сохраняется в объекте **BitMap** методом **Assign**.

11. В обработчик события формы **OnDestroy** запишите оператор

```
BitMap->Free();
```

который освобождает память при закрытии приложения.

12. Для подраздела меню Открыть в обработчик включите операторы:

```

if (OpenPictureDialog1->Execute()) {
    Image3->Picture->LoadFromFile(OpenPictureDialog1->FileName);
    BitMap->Assign(Image3->Picture);
}

```

Эти операторы загружают в компонент **Image3** файл изображения, который выбирает в диалоге пользователь, и запоминают изображение в **BitMap**.

13. Для подраздела меню Отменить в обработчик включите оператор:

```
Image3->Picture->Assign(BitMap);
```

Этот оператор восстанавливает на холсте изображение, сохраненное в **BitMap**.

14. В обработчик события **OnClick** кнопок **SBBrush** и **SBColor** запишите оператор

```

if (((TSpeedButton *)Sender)->Down)
    BitMap->Assign(Image3->Picture);

```

Этот оператор запоминает в **BitMap** текущий вид изображения перед началом работы с очередным инструментом.

15. В обработчик события **OnMouseDown** компонентов **Image3** и **Image4** вставьте код:

```

if((Sender == Image4) || SBColor->Down)
// режим установки основного и вспомогательного цветов
{
    if(Button == mbLeft)
    {
        // установка основного цвета
        Image1->Canvas->Brush->Color =
            ((TImage *)Sender)->Canvas->Pixels[X][Y];
        Image1->Canvas->FillRect(Rect(0,0,Image1->Width,
            Image1->Height));
    }
    else
    {
        // установка вспомогательного цвета

```

```

Image2->Canvas->Brush->Color =
    ((TImage *)Sender)->Canvas->Pixels[X][Y];
Image2->Canvas->FillRect(Rect(0,0,Image2->Width,Image2->Height));
}
}
else if (SBBrush->Down)
// режим закраски указанной области холста
{
    if (Button==mbLeft)
        Image3->Canvas->Brush->Color = Image1->Canvas->Brush->Color;
    else Image3->Canvas->Brush->Color = Image2->Canvas->Brush->Color;
    Image3->Canvas->FloodFill(X,Y,
        Image3->Canvas->Pixels[X][Y],fsSurface);
}
}

```

Это основной код, осуществляющий как установку основного и вспомогательного цветов, так и функцию инструмента графического редактора — кисти. Если кнопка мыши нажата на палитре цветов **Image4** или если кнопка **SBColor** — кнопка указателя цвета утоплена, то приложение находится в режиме установки цветов. При нажатой левой кнопке мыши цвет пикселя под курсором мыши передается в окно основного цвета, а при нажатии правой кнопки — в окно вспомогательного цвета.

Если кнопка мыши нажата на холсте **Image3** и при этом кнопка **SBBrush** утоплена, то приложение находится в режиме закраски указанной области рисунка. В этом случае в зависимости от нажатой кнопки мыши выбирается основной или вспомогательный цвет и функцией **FloodFill** производится закраска области, координаты внутренней точки которой указаны курсором мыши, а цвет — цветом пикселя, на который указывает мышь.

Ваше приложение готово. В дальнейшем, после получения некоторых дополнительных сведений о графике, вы, если захотите, сможете вернуться к нему и пополнить новыми инструментами.

Откомпилируйте приложение, сохраните его и выполните. Проверьте установку цветов. При щелчке левой или правой кнопкой мыши на палитре цветов у вас будет меняться соответственно основной или вспомогательный цвет. Если вы нажмете кнопку кисти в окне приложения, то при щелчке на холсте у вас должна закрашиваться указанная курсором область в основной или вспомогательный цвет (рис. 5.12 а). Если вы отпустите кнопку кисти в окне приложения и нажмете кнопку определения цветов, то при щелчке на холсте у вас основной или вспомогательный цвет будет устанавливаться равным цвету холста под курсором мыши.

Если вы выполните команду Правка | Отменить, то изображение вернется к тому виду, какой был при начале работы с последним использованным инструментом.

Теперь вы можете загрузить командой Файл | Открыть вашего приложения какое-то изображение (на рис. 5.12 б загружен файл ...\\Images\\Splash\\16Color\\earth.bmp). Вы можете попробовать перекрашивать те или иные области изображения. Например, можете изменить цвет фона, выделить какую-то сплошную область рисунка одного цвета (на рис. 5.12 б выделена одна из областей облачного покрова) и т.п.

Итак, вы реализовали два из основных инструментов любого графического редактора рисунков: кисть и индикатор цвета. Реализация остальных типичных инструментов требует предварительного ознакомления с режимами рисования, что будет сделано в следующем разделе 5.1.5. После этого вы сможете, вероятно, при желании вернуться к вашему графическому редактору и попробовать реализовать, например, инструменты, рисующие различные фигуры и перемещающие фрагменты изображения.

5.1.5 Режимы рисования

5.1.5.1 Режимы пера

У пера **Pen** имеется еще одно свойство, которое мы пока не рассматривали. Это свойство — **Mode** (режим). Возможные значения **Mode** приведены в справочной части книги в главе 16. По умолчанию значение **Mode** = **pmCopy**. Это означает, что линии проводятся цветом, заданным в свойстве **Color**. Но возможны и другие режимы, в которых учитывается не только цвет **Color**, но и цвет соответствующих пикселей фона. Наиболее интересным из этих режимов является режим **pmNotXor** — сложение с фоном по инверсному исключающему ИЛИ. Операция инверсного исключающего ИЛИ анализирует по битам два своих операнда. Результирующий бит равен «0», если соответствующие биты двух операндов не равны друг другу, а при равенстве битов операндов результирующий бит равен «1».

Вспомните, что каждый пиксель хранит цвет как набор битов. Пусть, например, фоновый пиксель имеет значение 0110011, а цвет пера установлен в 1111000. Применение операции **pmNotXor** к этим двум числам даст цвет со значением 0110100. Этот цвет перо задаст данному пикселю. А теперь посмотрим, что получится, если перо повторно пройдет по тому же пикселю. В этом случае опять будет выполнена операция **pmNotXor** по отношению к цвету пера 1111000 и текущему цвету пикселя, который стал равен 0110100. Применение **pmNotXor** к этим числам даст в результате 0110011, т.е. первоначальный цвет пикселя.

Это значит, что если нарисовать на фоне какую-то фигуру один раз, а затем нарисовать ту же фигуру повторно, то нарисованная фигура исчезнет и каждый пиксель вернется к своему первоначальному цвету. Эту особенность режима **pmNotXor**, свойственную также режиму **pmXor** — сложение с фоном по исключающему ИЛИ, можно использовать для создания простенькой анимации. Достаточно нарисовать нечто, затем стереть нарисованное, перерисовать немного измененным — и рисунок будет представляться ожившим. В последующих разделах мы часто будем использовать такой режим рисования.

5.1.5.2 Режимы копирования и рисования канвы

Ранее мы рассматривали и использовали копирование одного графического объекта в другой методом **Assign**. Однако, у канвы имеются и другие методы копирования. Это прежде всего метод **CopyRect**, позволяющий копировать прямоугольную область источника изображения в прямоугольную область данной канвы. Метод определен следующим образом:

```
void __fastcall CopyRect(const Windows::TRect &Dest,  
                        TCanvas* Canvas,  
                        const Windows::TRect &Source);
```

Параметр **Dest** определяет прямоугольную область канвы, в которую производится копирование. Параметр **Canvas** указывает источник, из которого копируется изображение. Это может быть канва любого компонента: типа **TImage**, типа **TBitmap** и др. В частном случае источником может быть и канва того же компонента, в который производится копирование. Параметр **Source** определяет прямоугольную область в источнике изображения, которая копируется в область **Dest**. Обе прямоугольные области и в источнике, и в приемнике имеют тип **TRect**. В разделе 5.1.3.4 этот тип и работающая с ним функция **Rect** уже были рассмотрены.

Копирование — это не просто перенос изображения. В общем случае копирование означает сложное взаимодействие копируемого изображения и того, которое было до этого в области, куда производится копирование. Характер этого взаимодействия определяется параметром **CopyMode** (режим копирования) той канвы, в которую производится копирование. Вы можете в справочной части книги в главе 16 посмотреть 15 различных значений этого параметра. По умолчанию значение

CopyMode равно **cmSrcCopy**. Это единственный режим, при котором производится действительное копирование: изображение в **Dest** стирается и заменяется скопированным. Есть два значения — **cmWhiteness** и **cmBlackness**, при которых собственно никакое копирование не производится: просто область закрашивается соответственно белым или черным цветом. А все остальные режимы определяют сложное взаимодействие копируемого изображения с тем, которое было в **Dest**. Особый интерес представляет режим **cmSrcInvert**, при котором изображения канвы и источника комбинируются, используя булеву операцию XOR. Так же, как мы видели это для пера, повторное копирование в подобном режиме восстанавливает прежнее изображение на канве. Интересен также режим **cmSrcAnd**. Если копируемое изображение представляет собой контурный черный рисунок на белом фоне, то этот рисунок наложится на прежнее изображение канвы, а белый фон будет прозрачным, так что под ним будет видно первоначальное изображение. В режиме **cmSrcPaint** аналогичный эффект будет, если копируемое изображение представляет собой белый контурный рисунок на черном фоне.

Приведем примеры копирования в различных режимах. Операторы

```
Image1->Canvas->CopyMode = cmSrcCopy;
Image1->Canvas->CopyRect (Rect (0, 0, 200, 200), Image2->Canvas,
                           Rect (0, 0, 200, 200));
```

обеспечивают копирование изображения фрагмента канвы компонента **Image2** в указанную область канвы компонента **Image1**. Изображение, которое ранее было на канве компонента **Image1** в пределах области с координатами углов (0, 0) и (200, 200), просто заменяется новым.

Операторы

```
Image1->Canvas->CopyMode = cmSrcInvert;
Image1->Canvas->CopyRect (Rect (0, 0, 200, 200), Image2->Canvas,
                           Rect (0, 0, 200, 200));
...
Image1->Canvas->CopyRect (Rect (0, 0, 200, 200), Image2->Canvas,
                           Rect (0, 0, 200, 200));
```

обеспечивают копирование изображения фрагмента канвы компонента **Image2** в указанную область канвы компонента **Image1** в режиме **cmSrcInvert**. После выполнения функции **CopyRect** в первый раз изображения в компонентах **Image1** и **Image2** налагаются друг на друга, а в результате выполнения функции **CopyRect** во второй раз исходное изображение на канве компонента **Image1** восстанавливается.

Операторы

```
Image1->Canvas->CopyMode = cmWhiteness;
Image1->Canvas->CopyRect (Rect (0, 0, 200, 200), Image2->Canvas,
                           Rect (0, 0, 200, 200));
```

просто очищают указанную область канвы компонента **Image1**, закрашивая ее белым цветом. При этом изображение в компоненте **Image2** никак не участвует в операциях копирования.

На диске, прилагаемом к книге, имеется простой пример приложения, позволяющего наглядно посмотреть различные режимы копирования.

В приложении имеются два окна, содержащие разноцветные горизонтальные и вертикальные полосы. Соответствующими кнопками любое из них можно загрузить в третье окно, а затем скопировать в него изображение одного из первых двух окон. Предварительно в выпадающем списке можно выбрать режим копирования.

Поэкспериментируйте с этим приложением, чтобы понять различия между режимами копирования. Вы уже знаете все, чтобы самим спроектировать такое приложение.

Еще один метод копирования — **BrushCopy** вы можете посмотреть в справочной части книги. Он сохраняется, как заявляют сами авторы C++Builder, только для совместимости с более ранними версиями системы.

Помимо методов копирования свойство **CopyMode** используется также методом рисования на канве **Draw**. Его описание:

```
void __fastcall Draw(int X, int Y, TGraphic* Graphic);
```

Метод **Draw** рисует изображение, содержащееся в объекте, указанном параметром **Graphic**, сохраняя исходный размер изображения в его источнике и перенося изображение в область канвы объекта, верхний левый угол которой определяется параметрами **X** и **Y**. Источник изображения может быть битовой матрицей, пиктограммой или метафайлом. Если источник — объект типа **TBitmap**, то перенос изображения производится в режиме, установленном свойством канвы **CopyMode**.

Например, оператор

```
Image1->Canvas->Draw(10,10,Bitmap1);
```

рисует на канве компонента **Image1** изображение из компонента **Bitmap1** в область с координатами левого верхнего угла (10, 10).

Еще один метод рисования — **DrawFocusRect**. Этот метод рисует изображение прямоугольника в виде, используемом для отображения рамки фокуса, операцией XOR. Функция **DrawFocusRect** объявлена следующим образом:

```
void __fastcall DrawFocusRect(const Windows::TRect &Rect);
```

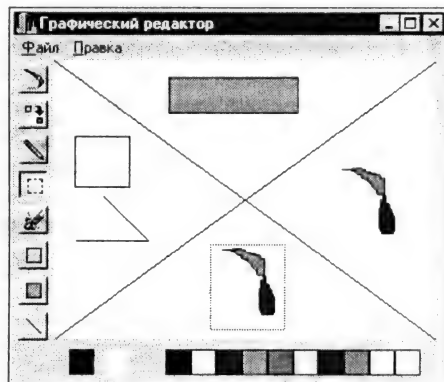
где **Rect** — прямоугольная область. Поскольку при рисовании используется операция XOR, то повторный вызов этого метода с тем же значением **Rect** удаляет изображение прямоугольника.

5.1.6 Продолжение создания собственного графического редактора

Теперь мы рассмотрели большинство вопросов, связанных с графикой. Можно вернуться к созданному нами ранее собственному графическому редактору (раздел 5.1.4) и попробовать его усовершенствовать. Подобный усовершенствованный редактор вы можете найти на диске, прилагаемом к книге. Его вид после выполнения ряда операций и в момент перетаскивания фрагмента изображения приведен на рис. 5.13. Редактор этот чисто демонстрационный и поэтому в нем нет многого, что должно быть в настоящем графическом редакторе. К тому же, и тот инструментарий, который в нем имеется, надо бы было реализовать компактнее и с большей надежностью. Но в данном случае, как и во всех демонстрационных примерах, функциональность принесена в жертву простоте и наглядности кода.

Рис. 5.13

Графический редактор в момент перетаскивания фрагмента изображения



Тем не менее, мы не будем подробно анализировать текст этого приложения. Рассмотрим только основные приемы, которые использованы при создании различных инструментов. А общую логику работы вы можете придумать сами или позаимствовать ее из примера.

Приложение выполняет следующие функции:

- Установка основного и дополнительного цветов. Щелчок на панели цветов левой кнопкой мыши устанавливает основной цвет, а щелчок правой кнопкой — вспомогательный.
- Кисть — кнопка **SBBrush**. Закрашивает замкнутую область, ограниченную цветом того пикселя, который указан щелчком мыши. При щелчке левой кнопкой закрашивание производится основным цветом, при щелчке правой кнопкой — вспомогательным.
- Индикация цвета — кнопка **SBColor**. В этом режиме вы можете указать курсором мыши любой пиксель на изображении и, щелкнув левой кнопкой, установить цвет этого пикселя как основной, а щелкнув правой кнопкой, установить его как вспомогательный цвет.
- Карандаш — кнопка **SBPen**. В этом режиме вы можете рисовать произвольную кривую основным цветом.
- Выделение фрагмента — кнопка **SBRect**. Фрагмент выделяется точечной рамкой. Выделенный фрагмент можно в дальнейшем перетащить мышью на другое место. Если в процессе перетаскивания нажата клавиша **Ctrl**, то производится копирование фрагмента, в противном случае — вырезание, при котором область начального размещения фрагмента закрашивается вспомогательным цветом. Выделенный фрагмент может быть также скопирован или вырезан в буфер обмена **Clipboard** соответствующими командами меню.
- Стирание изображения (ластик) — кнопка **SBErase**. Перемещение ластика закрашивает область под ним во вспомогательный цвет.
- Рисование прямоугольника — кнопка **SBRectang**. Рисуются прямоугольная рамка основным цветом.
- Рисование заполненного прямоугольника — кнопка **SBFillRec**. Рисуются прямоугольная рамка основным цветом и прямоугольник внутри закрашивается вспомогательным цветом.
- Рисование прямой линии — кнопка **SBLine**. Рисуются прямая линия основным цветом.
- Открытие графического файла — команда **Файл | Открыть**.
- Сохранение изображения в графическом файле — команда **Файл | Сохранить как**.
- Отмена операций, выполненных последним использованным инструментом — команда **Правка | Отменить**.
- Копирование или вырезание выделенного фрагмента изображения в буфер обмена **Clipboard** — команды **Правка | Копировать** или **Правка | Вырезать**.
- Вставка графического изображения типа битовой матрицы из буфера обмена **Clipboard** — команда **Правка | Вставить**.

Начнем с рассмотрения функции выделения фрагмента. Она осуществляется методом **DrawFocusRect**. В этом режиме при событии **OnMouseDown** холста — компонента **Image3**, выполняются операторы:

```
// Запоминание начального положения курсора мыши
X0 = X;
Y0 = Y;
```

```
// формирование начального положения области фрагмента
R.Top = X;
```

```

R.Bottom = X;
R.Left = Y;
R.Right = Y;

// Рисование рамки
Image3->Canvas->DrawFocusRect(R);
RBegin = true;

```

Эти операторы запоминают координаты мыши **X** и **Y** в переменных **X0** и **Y0**, задают начальные координаты прямоугольной области — переменной **R** типа **TRect** и рисуют рамку (пока нулевого размера) методом **DrawFocusRect**. Устанавливается также флаг начала выделения фрагмента — переменная **RBegin**.

При событии **OnMouseMove** компонента **Image3**, если установлен флаг **RBegin**, выполняются операторы:

```

// Стирание прежней рамки
Image3->Canvas->DrawFocusRect(R);

// формирование области R
if (X0 < X) { R.Left = X0; R.Right = X; }
else { R.Left = X; R.Right = X0; }
if (Y0 < Y) { R.Top = Y0; R.Bottom = Y; }
else { R.Top = Y; R.Bottom = Y0; }

// Рисование новой рамки
Image3->Canvas->DrawFocusRect(R);

```

Первый из этих операторов стирает прежнее изображение рамки (напомним, что метод **DrawFocusRect** рисует рамку с помощью операции **XOR**). Два следующих оператора формируют новую область **R** из начальных координат (**X0**, **Y0**) и текущих координат курсора (**X**, **Y**). Дело в том, что область, передаваемая в функцию **DrawFocusRect**, должна быть сформирована «правильно» — значение **R.Left** должно быть меньше **R.Right**, а **R.Top** — меньше **R.Bottom**. Поскольку пользователь может перемещать курсор в любом направлении и соотношение координат (**X0**, **Y0**) и (**X**, **Y**) может быть любым, требуется упорядочивание координат.

Последний оператор рисует рамку в новом положении.

Итак, рамка, ограничивающая фрагмент нарисована. Теперь рассмотрим процедуру перетаскивания пользователем выделенного фрагмента. Если пользователь помещает курсор внутрь выделенной области и нажимает кнопку мыши, выполняются операторы:

```

// Стирание прежней рамки
Image3->Canvas->DrawFocusRect(R);

// Установка флага перетаскивания
RDrag = true;

// Запоминание начального положения курсора мыши
X0 = X;
Y0 = Y;
// Запоминание начального положения перетаскиваемого фрагмента
R0 = R;

// Запоминание изображения
BitMap->Assign(Image3->Picture);

// Установка цвета кисти
Image3->Canvas->Brush->Color = Image2->Canvas->Brush->Color;

```

Первый оператор стирает рамку. Второй — устанавливает флаг перетаскивания — переменную **RDrag**. Два следующих оператора запоминают начальное положение перетаскиваемого фрагмента в переменной **R0** типа **TRect**. Следующий опе-

ратор запоминает методом **Assign** изображение в момент начала перетаскивания в переменной **Bitmap**. Это необходимо, чтобы в процессе перетаскивания можно было восстанавливать испорченные места изображения и чтобы при желании пользователя можно было в дальнейшем отменить результат перетаскивания. Последний оператор задает цвет кисти равным вспомогательному цвету, хранящемуся в компоненте **Image2**.

При событии **OnMouseMove** компонента **Image3**, если установлен флаг **RDrag**, выполняются операторы:

```
// Восстановление изображения под перетаскиваемым фрагментом
Image3->Canvas->CopyRect(R, BitMap->Canvas, R);

// Если не нажата клавиша Ctrl - стирание изображения в R0
if (! Shift.Contains(ssCtrl))
    Image3->Canvas->FillRect(R0);

// Формирование нового положения фрагмента
R.Left = R.Left + X - X0;
R.Right = R.Right + X - X0;
R.Top = R.Top + Y - Y0;
R.Bottom = R.Bottom + Y - Y0;

// Запоминание положения курсора мыши
X0 = X;
Y0 = Y;

// Рисование фрагмента в новом положении
Image3->Canvas->CopyRect(R, BitMap->Canvas, R0);

// Рисование рамки
Image3->Canvas->DrawFocusRect(R);
```

Первый оператор восстанавливает изображение под перетаскиваемым фрагментом в его прежней позиции (т.е. стирает фрагмент), копируя соответствующую область методом **CopyRect** из компонента **BitMap**. Далее, если не нажата клавиша **Ctrl**, то очищается область начального размещения фрагмента (осуществляется вырезание) методом **FillRect**. В противном случае начальное изображение фрагмента остается на месте. Затем запоминаются новые координаты курсора и новое положение фрагмента, после чего фрагмент и его рамка рисуются в новом положении.

Вот в общих чертах и все, что надо сделать для осуществления наиболее сложных операций — выделения фрагмента и его перетаскивания.

Режимы рисования заполненного и не заполненного прямоугольников проще. Начало этих режимов по событию **OnMouseDown** и их продолжение по событиям **OnMouseMove** не отличаются от рассмотренного ранее режима выделения фрагмента. Отличие только в том, что при завершении формирования пользователем прямоугольной рамки, т.е. при событии **OnMouseUp**, надо в данном случае нарисовать прямоугольник. Рисование заполненного прямоугольника осуществляется операторами:

```
Image3->Canvas->Brush->Color = Image2->Canvas->Brush->Color;
Image3->Canvas->Pen->Color = Image1->Canvas->Brush->Color;
Image3->Canvas->Rectangle(R.Left, R.Top, R.Right, R.Bottom);
```

Они задают цвета кисти и пера и рисуют прямоугольник методом **Rectangle**. Рисование не закрашенного прямоугольника осуществляется операторами:

```
Image3->Canvas->Brush->Color = Image1->Canvas->Brush->Color;
Image3->Canvas->FrameRect(R);
```

Обратите внимание, что равным основному цвету задается цвет кисти, а не пера, поскольку метод **FrameRect** рисует цветом кисти.

Рисование прямой линии осуществляется следующим образом. Начало рисования по событию **OnMouseDown** сводится к операторам:

```
X0 = X;  
Y0 = Y;  
X1 = X;  
Y1 = Y;  
Image3->Canvas->Pen->Mode = pmNotXor;  
Image3->Canvas->Pen->Color = Image1->Canvas->Brush->Color;
```

Они запоминают положение курсора в двух наборах переменных: **(X0,Y0)** и **(X1, Y1)**. Зачем нужны два набора — будет сказано позднее. Затем устанавливается цвет пера и режим **pmNotXor**, который позволит при движении мыши стирать изображение линии.

При событиях **OnMouseMove** работают следующие операторы:

```
// Стирание прежней линии  
Image3->Canvas->MoveTo (X0,Y0);  
Image3->Canvas->LineTo (X1,Y1);  
  
// Рисование новой линии  
Image3->Canvas->MoveTo (X0,Y0);  
Image3->Canvas->LineTo (X,Y);  
  
// Запоминание новых координат конца линии  
X1 = X;  
Y1 = Y;
```

В этих операциях сначала парой методов **MoveTo** и **LineTo** стирается линия в прежнем положении, а затем такой же парой методов рисуется новая линия. После этого запоминаются новые координаты конца линии.

Попробуем разобраться, зачем все так сложно. Ведь казалось бы, что достаточно всего двух операторов:

```
Image3->Canvas->LineTo (X0,Y0);  
Image3->Canvas->LineTo (X,Y);
```

Первый из них сотрет прежнюю линию, поскольку текущая позиция пера после рисования этой прежней линии соответствовала концу линии (тем координатам, которые мы храним в переменных **X1, Y1**). А второй оператор нарисует новую линию. И не надо хранить никаких координат.

Так просто нельзя сделать из-за одной тонкости: метод **LineTo** рисует линию, начинающуюся в текущей позиции пера и заканчивающуюся в указанной точке, исключая эту конечную точку. Поэтому, если выполнить в режиме **pmNotXor** приведенные выше операторы, то первый из них сотрет прежнюю линию, но оставит нестертой точку с координатами **(X0,Y0)** и кроме того нарисует точку в бывшей позиции курсора, поскольку она как конечная была исключена при рисовании предыдущей линии. Таким образом, на концах линии останется «грязь» в виде концевых точек, которая и будет тянуться за перемещающимся курсором мыши.

Заключительные операции при событии **OnMouseUp** аналогичны рассмотренным выше, но дополняются переводом пера в режим **pmCopy**, при котором рисуется окончательная линия:

```
// Стирание прежней линии  
Image3->Canvas->MoveTo (X0,Y0);  
Image3->Canvas->LineTo (X1,Y1);  
// Рисование новой линии  
Image3->Canvas->Pen->Mode = pmCopy;  
Image3->Canvas->MoveTo (X0,Y0);  
Image3->Canvas->LineTo (X,Y);
```

Теперь рассмотрим инструмент Перо, позволяющий рисовать произвольные линии. Казалось бы естественной реализацией этого инструмента был бы следующий оператор, включаемый в обработчик события **OnMouseMove**:

```
Image3->Canvas->Pixels[X][Y] = Image1->Canvas->Brush->Color;
```

который окрашивает пиксель под курсором в основной цвет. Однако, попробуйте так реализовать Перо и ничего хорошего не увидите. Курсор мыши перемещается быстро и события **OnMouseMove** происходят вовсе не при перемещении на соседний пиксель. Поэтому ваша линия распадется на отдельные точки, тем более редкие, чем быстрее пользователь будет перемещать курсор. Линию, оставляемую курсором, следует рисовать тоже методом **LineTo**, поместив в обработчик события **OnMouseMove** оператор

```
Image3->Canvas->LineTo(X,Y);
```

Мы рассмотрели почти все инструменты, введенные в приложении рис. 5.13. Коротко перечислим оставшееся. Ластик реализуется методом **FillRect**, очищающим изображение под его рамкой. Кисть, отмена команд и загрузка внешнего файла рассматривались в разделе 5.1.4. Сохранение файла осуществляется с использованием компонента типа **SavePictureDialog** оператором

```
if (SavePictureDialog1->Execute())
{
    BitMap->Assign(Image3->Picture);
    BitMap->SaveToFile(SavePictureDialog1->FileName);
}
```

Изображение с холста сохраняется в компоненте **Bitmap**, из которого методом **SaveToFile** записывается в выбранный пользователем файл.

Команды меню Копировать и Вырезать осуществляются процедурой

```
void __fastcall TForm1::MCopyClick(TObject *Sender)
{
    // Стирание рамки
    Image3->Canvas->DrawFocusRect(R);

    // Создание временного объекта BMCopy
    Graphics::TBitmap *BMCopy = new Graphics::TBitmap;
    BMCopy->Width = R.Right - R.Left;
    BMCopy->Height = R.Bottom - R.Top;
    try
    {
        // Копирование фрагмента в BMCopy
        BMCopy->Canvas->CopyRect(Rect(0,0,BMCopy->Width,
                                   BMCopy->Height),Image3->Canvas,R);
        // Восстановление рамки
        Image3->Canvas->DrawFocusRect(R);

        // Копирование в Clipboard
        Clipboard()->Assign(BMCopy);
        if (Sender == MCut)
        {
            // Вырезание
            Image3->Canvas->Brush->Color = clWhite;
            Image3->Canvas->FillRect(R);
        }
    }
    finally
    {
        // Освобождение памяти
        BMCopy->Free();
    }
}
```

Копированию или вырезанию подлежит ранее выделенный пользователем объект, местоположение и размеры которого определяются переменной **R**. Поэтому сначала создается временный объект типа **TBitmap**, в который переносится копируемый фрагмент. Затем этот объект копируется в буфер обмена Clipboard. Благодаря разделу **__finally** память освобождается от временного объекта при любом исходе копирования: удачном или аварийном.

Для работы с буфером обмена используется функция **Clipboard**, создающая объект типа **TClipboard**, инкапсулирующий свойства буфера обмена Windows.

Аналогично реализуется команда Вставить, копирующая изображение из буфера обмена Clipboard:

```
void __fastcall TForm1::MPasteClick(TObject *Sender)
{
    Graphics::TBitmap *BMCopy = new Graphics::TBitmap;
    try
    {
        try
        {
            BMCopy->LoadFromClipboardFormat(CF_BITMAP,
                Clipboard()->GetAsHandle(CF_BITMAP), 0);
            Image3->Canvas->CopyRect(Rect(0, 0, BMCopy->Width,
                BMCopy->Height), BMCopy->Canvas,
                Rect(0, 0, BMCopy->Width,
                BMCopy->Height));
        }
        __finally
        {
            BMCopy->Free();
        }
    }
    catch (EInvalidGraphic&)
    {
        ShowMessage("Ошибочный формат графики");
    }
}
```

Чтение из Clipboard осуществляется методом **LoadFromClipboardFormat**. Предусмотрен перехват исключения **EInvalidGraphic**, если в Clipboard содержится не битовая матрица.

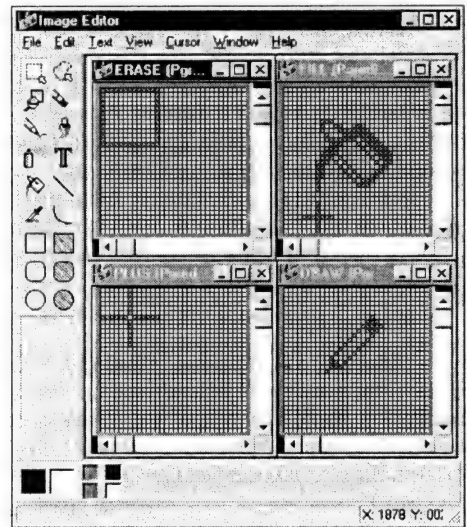
Попробуйте реализовать описанный графический редактор или разберитесь подробнее в его работе, посмотрев его код на прилагаемом к книге диске. Попробуйте усовершенствовать редактор, добавив, в него, например, выбор ширины линий, рисование эллипсов и т.д. Вы можете также усовершенствовать ваш графический редактор, введя в него различные курсоры в зависимости от выбранного пользователем инструмента. Методика создания собственных курсоров и включения их в файл ресурсов приложения описана в разделах 5.1.2.4 и 4.5.6. На рис. 5.14 показаны курсоры, подготовленные для включения в графический редактор. Изображения вы можете нарисовать сами или почерпнуть их из примера **Examples\Apps\Doodle\extrares.res**, поставляемого с C++Builder 5. Чтобы перенести изображения курсоров из одного файла в другой, вы можете открыть два Редактора Изображений и передать рисунки из одного редактора в другой через буфер обмена.

Если вы включили таким образом курсоры в файл ресурсов, то использовать их в приложении можно следующим образом (см. раздел 4.5.6). Надо объявить глобальные переменные ваших курсоров:

```
// константы курсоров
const int crFill = 1;
const int crPlus = 2;
const int crDraw = 3;
const int crErase = 4;
```

Рис. 5.14

Курсоры для графического редактора



В событии формы **OnCreate** надо зарегистрировать курсоры операторами:

```
// регистрация курсоров
Screen->Cursors[crFill] = LoadCursor(HInstance, «FILL»);
Screen->Cursors[crPlus] = LoadCursor(HInstance, «PLUS»);
Screen->Cursors[crDraw] = LoadCursor(HInstance, «DRAW»);
Screen->Cursors[crErase] = LoadCursor(HInstance, «ERASE»);
```

А в общем обработчике событий **OnClick** инструментальных кнопок ввести следующий код, задающий для компонента **Image3** тот или иной вид курсора в зависимости от нажатой кнопки:

```
if((Sender == SBPen)||(Sender == SBLine))
    Image3->Cursor = TCursor(crDraw);
else if(Sender == SBBrush)
    Image3->Cursor = TCursor(crFill);
else if(Sender == SBErase)
    Image3->Cursor = TCursor(crErase);
else if(Sender == SBColor)
    Image3->Cursor = TCursor(crDefault);
else Image3->Cursor = TCursor(crPlus);
```

5.1.7 События OnPaint

До сих пор мы рисовали в основном на канве компонента **Image**. Но канву имеет не только **Image**. Ее имеют и многие другие компоненты, например, формы. Все, что ранее вы рисовали на канве компонентов типа **TImage**, вы могли бы рисовать и на форме. Кроме того имеется специальный компонент **PaintBox**, имеющий канву и позволяющий рисовать на ней. Рисование на **PaintBox** вместо формы не имеет никаких преимуществ, кроме, может быть, некоторого облегчения в расположении одного или нескольких рисунков в площади окна.

При рисовании на канве формы или **PaintBox** надо учитывать некоторые особенности. Давайте сначала выясним на собственном опыте, о чем идет речь.

Откройте новое приложение, перенесите на него диалог **OpenPictureDialog**, кнопку и в обработчик щелчка на ней вставьте операторы:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    if (OpenPictureDialog1->Execute())
```

```

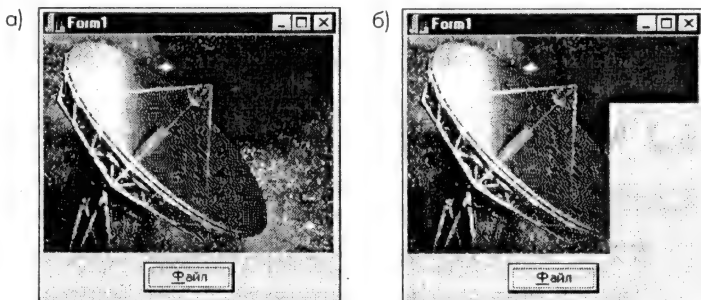
{
    Graphics::TBitmap *Bitmap = new Graphics::TBitmap;
    try
    {
        Bitmap->LoadFromFile (OpenPictureDialog1->FileName);
        Canvas->Draw(0,0,Bitmap);
    }
    finally
    {
        Bitmap->Free();
    }
}
}

```

Эти операторы обеспечивают загрузку выбранного пользователем графического файла и отображение изображения непосредственно на канве формы (поскольку оператор **Canvas->Draw** относится к канве формы, можно было бы это уточнить, написав **Form1->Canvas->Draw**). Запустите приложение, выберите файл и вы увидите что-нибудь вроде представленного на рис. 5.15 а. А теперь, не закрывая своего приложения, вернитесь в C++Builder и, ничего там не делая, опять перейдите в свое выполняющееся приложение. Если окно Редактора Кода, выступившее на первый план при вашем переходе в C++Builder, целиком перекрыло окно вашего приложения, то вернувшись в него вы увидите, что картинка в окне исчезла. Если же вы опять загрузите в него картинку и сдвинете окно приложения так, чтобы окно Редактора Кода не могло целиком его закрыть, то, повторив эксперимент с переходом в C++Builder и обратно вы, возможно, увидите результат, подобный представленному на рис. 5.15 б.

Рис. 5.15

Демонстрация исходного изображения (а) и его стирания (б) при перекрытии его другим окном



Вы видите, что если окно какого-то другого приложения перекрывает на время окно вашего приложения, то изображение, нарисованное на канве формы, портится. В компоненте **Image** этого не происходило, поскольку в классе **TImage** уже предусмотрены все необходимые действия, осуществляющие перерисовку испорченного изображения. А при рисовании на канве формы или других оконных компонентов эти меры должен принимать сам разработчик приложения.

Если окно было перекрыто и изображение испортилось, операционная система сообщает приложению, что в окружении что-то изменилось и что приложение должно предпринять соответствующие действия. Как только требуется обновление окна, для него генерируется событие **OnPaint**. В обработчике этого события (в нашем случае события формы) нужно перерисовать изображение.

Перерисовка может производиться разными способами в зависимости от приложения. В нашем примере можно было бы вынести объявление указателя **Bitmap** за пределы приведенной выше процедуры, т.е. сделать эту переменную глобальной:

```
Graphics::TBitmap *Bitmap;
```

Тогда приведенная выше процедура загрузки файла сокращается до

```
if (OpenPictureDialog1->Execute())
{
    Bitmap = new Graphics::TBitmap;
    Bitmap->LoadFromFile(OpenPictureDialog1->FileName);
    Canvas->Draw(0,0,Bitmap);
}
```

Оператор **Bitmap->Free()**, содержащийся ранее в этой процедуре, переносится в обработчик события формы **OnDestroy**. Тогда в течение всего времени выполнения вашего приложения вы будете иметь копию картинку в компоненте **Bitmap** и вам достаточно ввести в обработчик события **OnPaint** формы всего один оператор:

```
if (Bitmap != NULL) Canvas->Draw(0,0,Bitmap);
```

Оператор **if** используется, чтобы избежать ошибочного обращения к **Bitmap**, пока графический файл еще не загружался и объект **Bitmap** не создан.

Сделайте это, и увидите, что изображение на форме не портится при любых перекрытиях окон.

Сделанный вами обработчик перерисовывает все изображение, хотя, может быть, испорчена только часть его. При больших изображениях это может существенно замедлять перерисовку и вызывать неприятные зрительные эффекты. Перерисовку можно существенно ускорить, если перерисовывать только испорченную область канвы. У канвы есть свойство **ClipRect** типа **TRect**, которое в момент обработки события **OnPaint** указывает область, которая подлежит перерисовке. Поэтому более экономным будет обработчик:

```
if (Bitmap != NULL)
    Canvas->CopyRect(Canvas->ClipRect, Bitmap->Canvas,
                    Canvas->ClipRect);
```

Он перерисовывает только область **ClipRect**, которая испорчена.

5.2 Мультимедиа и анимация

5.2.1 Звук

5.2.1.1 Типы звуковых и мультимедиа файлов

Так же, как существует множество рассмотренных нами форматов графических файлов, существует немало файлов звуковых и мультимедиа. Мы коротко охарактеризуем обе эти группы файлов в рамках данного раздела, так как файлы мультимедиа часто содержат звуковую дорожку и было бы не очень правомерно говорить о звуке не упомянуть звук в мультимедиа.

Наиболее простым звуковым файлом является волновой файл **.wav**. В нем записано цифровое представление информации о волновой форме электрического сигнала, соответствующего каждому звуку. Волновой файл «не знает» вообще ничего о том, что такое звук и что он означает; поэтому для хранения звукового клипа приходится запоминать массу информации.

Другим часто применяемым типом файлов-носителей являются файлы *цифрового интерфейса музыкальных инструментов (MIDI)*. Файлы **.midi** используются для хранения музыкальных фрагментов. В этих файлах звук хранится в виде данных о том, на каких инструментах исполняются определенные ноты и как долго они звучат. Если вы знакомы с музыкой, то можете рассматривать содержимое такого файла как цифровой эквивалент дирижерской партитуры. Одним из главных преимуществ MIDI является то, что файлы получают сравнительно небольшими. Файлы MIDI относятся к волновым файлам примерно так же, как метафайлы — к файлам **.bmp**. В обоих случаях файлы первого типа «понимают», какие данные

они представляют, а файлы второго типа хранят сырые данные, просто посылаемые на выходное устройство.

Волновые и MIDI файлы могут хранить только звук или музыку. Для хранения видео информации разработан ряд форматов. Отметим среди них файлы AVI и MPEG. Большинство видеофайлов поддерживают также хранение звуковой дорожки, так что звук воспроизводится синхронно с картинкой.

Что собой представляет видеофайл, и как он работает? Человеческий мозг интерпретирует быструю последовательность изображений, незначительно отличающихся друг от друга, как движение. Каждое из этих изображений называется *кадром*. Каждый следующий кадр несколько отличен от предыдущего. Чтобы мозг воспринимал смену кадров как плавное движение, желательно воспроизводить около 30 кадров в секунду. Более высокая частота не приводит к заметному росту качества, а более низкая производит впечатление мерцания экрана.

Если бы каждый кадр хранился в файле в виде битовой матрицы экрана (а это несколько сотен килобайт), то потребовался бы огромный объем дисковой памяти. При такой простой схеме хранения на компакт-диск, например, можно было бы записать всего 72 секунды видеофильма. Реально для хранения видеофильмов используется техника сжатия видеоданных.

Если не углубляться в сложную математику методов сжатия, то суть сводится к следующему. Когда захватывается очередной кадр, аппаратура или программа сжатия задается вопросом: «Можно ли сохранить этот кадр более компактно, если записать только то, что в нем отличается от предыдущего, или нужно сохранить картинку целиком?» Чаще всего выгодно сохранять только изменившиеся части сцены. Но в определенных обстоятельствах, например, при переключении на другую камеру, накладные расходы описания изменений заняли бы больше места, чем непосредственное сохранение кадра.

В методах хранения мультимедиа достигнуты большие успехи. Сейчас можно записать целый полнометражный кинофильм на стандартном CD-ROM.

5.2.1.2 Процедуры воспроизведения звуков Beep, MessageBeep и PlaySound

Наиболее простой процедурой, управляющей звуком, является процедура **Beep**. Она не имеет параметров и воспроизводит стандартный звуковой сигнал, установленный в Windows, если компьютер имеет звуковую карту и стандартный сигнал задан (он устанавливается в программе Windows «Панель управления» после щелчка на пиктограмме Звук). Если звуковой карты нет или стандартный сигнал не установлен, звук воспроизводится через динамик компьютера просто в виде короткого щелчка.

Откройте новое приложение, введите в него кнопку, в обработчике щелчка которой напишите одно слово:

```
Beep();
```

Можете запустить приложение, щелкнуть на кнопке и прослушать стандартный звук Windows или просто щелчок, если стандартный звук не установлен.

Более серьезной процедурой является функция Windows API **MessageBeep**:

```
bool MessageBeep(int uType);
```

Параметр **uType** указывает воспроизводимый звук как идентификатор раздела [sounds] реестра, в котором записаны звуки, сопровождающие те или иные события в Windows. С помощью приложения Звук в «Панели управления» пользователь может удалить или установит соответствующие звуки.

Параметр **uType** может иметь следующие значения:

Значение	Звук
MB_ICONASTERISK	SystemAsterisk — звездочка
MB_ICONEXCLAMATION	SystemExclamation — восклицание
MB_ICONHAND	SystemHand — критическая ошибка
MB_ICONQUESTION	SystemQuestion — вопрос
MB_OK	SystemDefault — стандартный звук

После запроса звука функция **MessageBeep** возвращает управление вызвавшей функции и воспроизводит звук асинхронно. Во время воспроизведения приложение может продолжать выполняться.

Если невозможно воспроизвести указанный в функции звук, делается попытка воспроизвести стандартный системный звук, установленный по умолчанию. Если и это невозможно, то воспроизводится стандартный сигнал через динамик.

При успешном выполнении возвращается ненулевое значение. При аварийном завершении возвращается ноль.

Можете в своем тестовом приложении ввести еще одну кнопку и написать для нее обработчик:

```
MessageBeep (MB_OK) ;
```

Вы услышите тот же стандартный звук Windows, что и при выполнении процедуры **Beep**. Или услышите тот же тихий щелчок, если стандартный звук не установлен. Попробуйте установить различные звуки с помощью «Панели управления» и проверить **MessageBeep** при различных значениях ее параметра.

А теперь давайте займемся более серьезной функцией **PlaySound**, которая позволяет воспроизводить не только звуки событий Windows, но и любые волновые файлы. Это функция API Windows, параметры которой описаны в модуле **mmsystem**. Поэтому для использования этой функции в вашем приложении необходимо включить директиву **#include <mmsystem.cpp>**, поскольку автоматически C++Builder ее не включает.

Функция **PlaySound** определена следующим образом:

```
bool PlaySound(char * pszSound, HINST hmod, int fdwSound);
```

Параметр **pszSound** представляет собой строку с нулевым символом в конце и определяет воспроизводимый звук. Параметр **hmod** используется, если звук берется из ресурса. А поскольку далее звуком из ресурса мы пользоваться не будем, то **hmod** всегда можно задавать равным 0.

Параметр **fdwSound** является множеством флагов, которые определяют режим воспроизведения и тип источника звука. Полное описание возможных значений флагов вы можете найти в главе 15 в разделе 15.7.3. Ниже приведены только некоторые из них, но наиболее важные для воспроизведения произвольных волновых файлов:

SND_ASYNC	Звук воспроизводится асинхронно и функция PlaySound возвращается немедленно после начала воспроизведения. Чтобы прекратить асинхронное воспроизведение волнового файла, надо вызвать PlaySound с параметром pszSound , равным 0
SND_LOOP	Воспроизведение звука постоянно повторяется, пока не вызовется PlaySound с параметром pszSound , равным 0. Одновременно надо установить флаг SND_ASYNC асинхронного воспроизведения звука

SND_NOSTOP	Если заданный звук не может быть воспроизведен, поскольку ресурсы, необходимые для воспроизведения, заняты воспроизведением другого звука, функция PlaySound немедленно вернет false , не воспроизводя заданного звука. Если данный флаг не указан, функция PlaySound пытается остановить воспроизведение другого звука, чтобы устройство могло быть использовано для воспроизведения нового звука
SND_NOWAIT	Если драйвер занят, функция сразу вернется без воспроизведения заданного звука
SND_PURGE	Останавливается воспроизведение любых звуков, вызванных в данной задаче. Если pszSound не 0, останавливаются все экземпляры указанного звука. Если pszSound равен 0, то останавливаются все звуки, связанные с данной задачей
SND_SYNC	Синхронное воспроизведение звука события. Функция PlaySound возвращается только после окончания воспроизведения

Флаги могут комбинироваться операцией **or**.

Звук, указанный параметром **pszSound**, должен помещаться в доступную память и должен подходить для установленного драйвера устройства воспроизведения волновых файлов. Функция **PlaySound** ищет файл звука в следующих каталогах: текущем, каталоге Windows, системном каталоге Windows, каталогах, перечисленных в переменной среды PATH, в списке каталогов, предоставляемых сетью. Если указанный звук не находится, функция **PlaySound** воспроизводит системный звук по умолчанию. Если функция не может найти и его, то воспроизведения не будет, а вернется значение **false**.

Приведем примеры использования функции **PlaySound**.

Оператор

```
PlaySound("C:\\Windows\\Media\\Звук Microsoft.wav", 0, SND_ASYNC);
```

воспроизводит асинхронно и однократно стандартный звук Microsoft, который вы обычно можете слышать при открытии Windows. В процессе воспроизведения продолжается выполнение приложения.

Чтобы опробовать функцию **PlaySound**, введите в свое приложение диалог **OpenDialog** и кнопку со следующим обработчиком щелчка:

```
if (OpenDialog1->Execute())  
    PlaySound(OpenDialog1->FileName.c_str(), 0, SND_ASYNC);
```

Запустите приложение, выберите файл какой-нибудь приятной музыки и работайте со своим приложением, наслаждаясь попутно выбранной мелодией.

В предыдущих примерах звук задавался именем его волнового файла. Функция **PlaySound** позволяет воспроизводить и системные звуки, просто называя их псевдонимы. Например, оператор

```
PlaySound("SystemStart", 0, SND_ASYNC);
```

воспроизведет тот же звук открытия Windows, что и приведенный ранее оператор, указывавший имя и путь к нему.

Оператор

```
PlaySound("C:\\Windows\\Media\\Звук Microsoft.wav", 0,  
    SND_ASYNC | SND_LOOP);
```

многократно асинхронно воспроизводит стандартный звук Microsoft, начиная его снова и снова, как только он заканчивается. Если вы ввели в свое приложение подобный оператор (пусть даже и с очень приятной музыкой), вам надо предусмотреть

реть еще и какую-нибудь кнопку, по которой воспроизведение прерывается заданием нового звука или выполнением оператора

```
PlaySound(0,0, SND_PURGE);
```

Полезно также ввести аналогичный оператор в обработчик события формы **OnCloseQuery**, так как звучание будет бесконечным и при работе с некоторыми версиями Windows у вас могут возникнуть проблемы при попытке закрыть ваше приложение. Кстати, подобный оператор в обработчике **OnCloseQuery** полезен в этих случаях не только при заказе бесконечного воспроизведения, но и в других случаях, когда вы хотите прервать воспроизведение с окончанием работы приложения.

Оператор

```
PlaySound("C:\\Windows\\Media\\Звук Microsoft.wav",0,SND_SYNC);
```

будет воспроизводить звук синхронно. Т.е. функция **PlaySound** не вернется, пока воспроизведение не завершится. На это время ваше приложение будет заблокировано. Впрочем, ваши действия во время этого вынужденного простоя запомнятся системой. И если вы, слушая музыку, щелкнули на той же кнопке повторно, то после окончания звука он будет повторен, так как ваш щелчок встал в очередь событий.

Все рассмотренные ранее операторы прерывали при своем выполнении звук, который асинхронно воспроизводился в момент вызова **PlaySound**. Если же вы выполните оператор с флагом **SND_NOSTOP**, например:

```
PlaySound("C:\\Windows\\Media\\Звук Microsoft.wav",0,  
SND_SYNC | SND_NOSTOP)
```

то в случае, если в этот момент драйвер занят воспроизведением другого звука, это воспроизведение не будет прерываться, а функция **PlaySound** сразу вернет **false**. Заказанного этим оператором звука вы не услышите, т.к. в очередь он не встанет.

Мы рассмотрели основные функции воспроизведения звука. В C++Builder имеется также компонент **MediaPlayer** — универсальный проигрыватель аудио- и видео-информации. Он будет рассмотрен в разделе 5.2.4.

5.2.2 Начала анимации — создание собственной мультипликации

А теперь давайте займемся движущимися изображениями. Но прежде, чем учиться просматривать готовые видеофайлы, попробуем свои силы в создании собственной анимации.

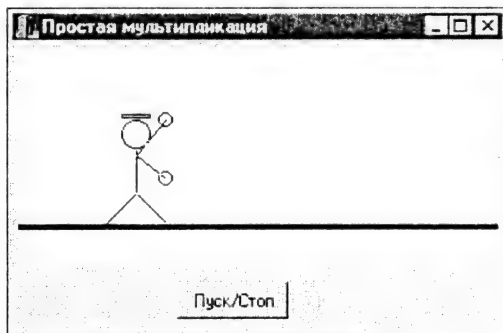
Каждый примерно представляет себе принципы создания мультипликационных фильмов, знает, что они представляют собой совокупность множества кадров, каждый из которых чуть-чуть отличается от предыдущего. Это при быстром поочередном просмотре кадров и создает иллюзию движения. Вам, конечно, в своей работе не придется рисовать с помощью C++Builder мультфильмы. Для этого имеются совершенно другие инструменты. Но некоторые простенькие анимации — оживление изображений, иногда желательно делать. Например, при создании какой-нибудь обучающей программы может захотеться оживить какие-то схемы или условные изображения механизмов, чтобы показать в движении взаимодействие их отдельных составляющих. Или применить анимации типа тех, которые используются в программе Windows «Проводник» при копировании и удалении файлов.

Давайте попробуем сделать простую мультипликацию. Но, чтобы не связываться с изображениями каких-то механизмов, сделаем нечто всем понятное: например условное изображение человечка, шагающего и бьющего при этом в литавры.

Откройте новое приложение. Перенесите на форму компоненты **Image**, кнопку **Button** и таймер **Timer**. Кнопку разместите внизу формы. Основную площадь формы должен занимать компонент **Image**, на котором и будет рисоваться изображение (рис. 5.16).

Рис. 5.16

Приложение с простой мультипликацией



Таймер будет задавать темп смены кадров. Поскольку в первом варианте приложения у нас будет всего два кадра, задайте значение свойства **Interval** таймера достаточно большим, например, 500 (поскольку интервалы задаются в миллисекундах, то это значение соответствует 0,5 сек). Значение параметра **Enabled** таймера установите в **false**. Таймер у нас будет управляться кнопкой.

Теперь размещение компонентов закончено. Надо ввести текст программы. В заголовочном файле добавьте строку

```
void __fastcall Draw();
```

Это объявление функции, которая будет рисовать изображение. А текст самого модуля может иметь вид:

```
short int num = 0;
short int H=20;           // шаг
short int Xpos = 2 * H;   // координата туловища
short int Ypos = 120;     // «земля»
short int Hmen = 30;      // высота тела
short int Rhead = 10;     // радиус головы
short int Rhead2 = Rhead / 2; // радиус литавров
short int revers = 1;      // направление движения
short int L = H * 1.41;    // длина ноги

// _____
void __fastcall TForm1::Draw()
{
    short int Yhead;       // координата низа головы
    switch (num)
    {
    case 0:
        Yhead = Ypos-H-Hmen;
        Image1->Canvas->MoveTo(Xpos-H, Ypos);
        Image1->Canvas->LineTo(Xpos, Ypos-H);
        Image1->Canvas->LineTo(Xpos+H, Ypos); // другая нога
        Image1->Canvas->MoveTo(Xpos, Ypos-H);
        Image1->Canvas->LineTo(Xpos, Yhead); // туловище
        Image1->Canvas->MoveTo(Xpos+revers*H, Yhead-H);
        Image1->Canvas->LineTo(Xpos, Yhead+4); // рука
        Image1->Canvas->Ellipse(Xpos+revers*H-Rhead2, Yhead-H-Rhead2,
                               Xpos+revers*H+Rhead2, Yhead-H+Rhead2);
        Image1->Canvas->LineTo(Xpos+revers*H, Yhead+H); // другая рука
        Image1->Canvas->Ellipse(Xpos+revers*H-Rhead2, Yhead+H-Rhead2,
                               Xpos+revers*H+Rhead2, Yhead+H+Rhead2);
        Image1->Canvas->Ellipse(Xpos-Rhead, Yhead, Xpos+Rhead,
                               Yhead-2*Rhead);
        Image1->Canvas->Rectangle(Xpos-Rhead, Yhead-2*Rhead-1,
                                Xpos+Rhead, Yhead-2*Rhead-4); // шляпа
        break;
```

```

case 1:
    Yhead = Ypos-L-Hmen;
    Image1->Canvas->MoveTo(Xpos,Ypos);
    Image1->Canvas->LineTo(Xpos,Yhead);
    Image1->Canvas->MoveTo(Xpos,Yhead+4);
    Image1->Canvas->LineTo(Xpos+revers*L,Yhead+4);
    Image1->Canvas->Ellipse(Xpos+revers*L-Rhead2,Yhead+4-Rhead2,
                           Xpos+revers*L+Rhead2,Yhead+4+Rhead2);
    Image1->Canvas->Ellipse(Xpos-Rhead,Yhead,Xpos+Rhead,
                           Yhead-2*Rhead);
    Image1->Canvas->Rectangle(Xpos-H / 2,Yhead-2*Rhead-1,
                             Xpos+H / 2,Yhead-2*Rhead-4);
}
}
//-----
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    Draw();
    if ((Xpos >= Image1->Picture->Width-H) || (Xpos <= H))
        revers = -revers;
    Xpos = Xpos + revers * H;
    num = 1 - num;
    Draw();
}
//-----
void __fastcall TForm1::BRunClick(TObject *Sender)
{
    Timer1->Enabled = ! Timer1->Enabled;
}
//-----
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Image1->Canvas->MoveTo(0,Ypos+3);
    Image1->Canvas->Pen->Width = 4;
    Image1->Canvas->LineTo(Image1->ClientWidth,Ypos+3); // земля
    Image1->Canvas->Pen->Width = 1;
    Image1->Canvas->Pen->Mode = pmNotXor;
    Draw();
}

```

Начнем анализ этого кода с конца — с последней процедуры **FormCreate**, являющейся обработчиком события **OnCreate** формы. В этой процедуре рисуется линия, отображающая «землю», по которой будет ходить наш человечек. Затем устанавливается режим пера **pmNotXor**. И в заключение вызывается процедура **Draw**, которая рисует исходное положение человечка.

Процедура **BRunClick** является обработчиком события **OnClick** кнопки. Каждый щелчок на кнопке включает или выключает таймер, в результате чего человечек идет или останавливается.

Процедура **Timer1Timer** является обработчиком события **OnTimer** таймера. Это событие означает, что надо стереть прежний кадр и нарисовать новый. Сначала вызывается процедура **Draw**. Поскольку позиция человечка с момента показа предыдущего кадра не изменилась, то этот вызов рисует на том же самом месте, на котором рисовался предыдущий кадр. Следовательно, предыдущий рисунок стирается. Затем анализируется позиция человечка **Xpos**. Если эта позиция отстоит от какого-либо конца холста **Image1** на величину, меньшую шага **H**, то изменяется на обратный знак переменной **revers**, характеризующей направление движения. Если **revers = 1**, человечек шагает вправо; если **revers = -1**, человечек шагает влево. Затем позиция **Xpos** изменяется на величину **revers * H**, т.е. на шаг вправо или влево. Изменяется переменная **num**, которая указывает номер высвечиваемого

кадра: 0 или 1. В заключение вызывается процедура **Draw**, которая рисует указанный кадр в указанной позиции.

Последняя процедура, которую мы рассмотрим — процедура **Draw**, рисующая кадр. Она достаточно длинная, но в ней нет ничего сложного. В зависимости от значения **num** рисуется один или другой кадр, причем в рисунке учитывается позиция **Xpos** и направление движения **revers**.

Сохраните ваше приложение и выполните его. Щелкнув на кнопке вы можете заставить вашего человечка перемещаться. Достигнув края формы он будет поворачиваться и шагать в обратном направлении. При вторичном щелчке на кнопке он будет останавливаться.

Конечно, пока движения нашего человечка очень неуклюжи. Чуть позже мы научим его двигаться более плавно. А пока обсудим некоторые проблемы, связанные с построением даже простеньких мультипликаций.

Первая из них — создание фона. Наш человечек движется в пустом пространстве и мы не замечаем этой проблемы. Но попробуйте вставить в программу какой-нибудь фон. Например, вставьте в начало процедуры **FormCreate** следующие операторы, рисующие черный прямоугольник:

```
Image1->Canvas->Brush->Color = 0;  
Image1->Canvas->Rectangle(90,0,200,100);  
Image1->Canvas->Brush->Color = clWhite;
```

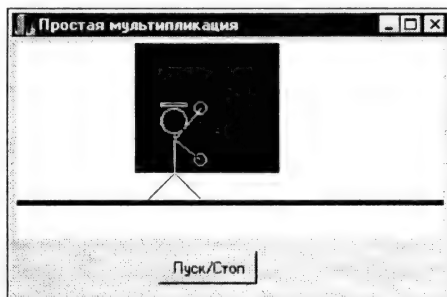
Выполните теперь свое приложение. Во время движения человечка вы увидите картину, приведенную на рис. 5.17. На черном фоне черный человечек становится белым. Если мультипликация черно-белая, то такой результат может только радовать, поскольку черное на черном просто исчезало бы. Но при разноцветном пестром фоне картина становится безрадостной. Вы можете в этом убедиться, если введете в свое приложение компонент **OpenPictureDialog** и в начале процедуры **FormCreate** вставите оператор

```
if (OpenPictureDialog1->Execute())  
    Image1->Picture->LoadFromFile(OpenPictureDialog1->FileName);
```

который позволит вам перед началом работы приложения загрузить в виде фона какой-нибудь графический файл, например, изображение земного шара, использованное ранее на рис. 5.1–5.3. Тогда вы увидите, что на пестром фоне земного шара ваш человечек будет совершенно теряться.

Рис. 5.17

Изменение цвета мультипликации при наложении на однотонный фон



Какие возможны выходы из положения? Наиболее простой — ограничиваться черно-белой мультипликацией или, во всяком случае, не использовать пестрых фонов. Если же по какой-то причине это невозможно, то, вероятнее всего, вам придется отказаться от режима пера **pmNotXor** и использовать буферизацию фона.

В нашем примере это можно было бы сделать следующим образом. Откройте свой предыдущий проект мультипликации и сохраните его под новым именем командой **File | Save Project As**. Затем сохраните под новым именем файл модуля коман-

дой File | Save As. Теперь вы можете вводить в модуль изменения, не опасаясь испортить свой предыдущий проект.

Введите глобальную переменную типа **TBitmap**:

```
Graphics::TBitmap *BitMap;
```

Это будет объект, в котором вы будете сохранять фрагменты фона, испорченные очередным кадром. Переделайте процедуру **FormCreate** следующим образом:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    BitMap = new Graphics::TBitmap;
    BitMap->Width = 2 * (L + Rhead);
    BitMap->Height = L + Hmen + 2 * Rhead + 6;
    Image1->Canvas->MoveTo(0, Ypos+3);
    Image1->Canvas->Pen->Width = 4;
    Image1->Canvas->LineTo(Image1->ClientWidth, Ypos+3); // земля
    Image1->Canvas->Pen->Width = 2;
    BitMap->Canvas->CopyRect(Rect(0,0, BitMap->Width,
                                BitMap->Height),
        Image1->Canvas, Rect(Xpos-L-Rhead, Ypos-(L+Hmen+2*Rhead+5),
                                Xpos+L+Rhead, Ypos+1));
    Draw();
}
```

Первыми операторами этой процедуры вы создаете объект **BitMap** и задаете его размеры равными максимальным размерам изображения человечка. В конце процедуры, перед вызовом **Draw** в компонент **BitMap** методом **CopyRect** копируется фрагмент изображения, внутри которого будет расположен рисунок человечка. После этого процедурой **Draw** рисуется соответствующий кадр. Обратите внимание, что в данном приложении отсутствует оператор, задававший ранее режим пера **pmNotXor**. Так что по умолчанию рисунок будет делаться обычным образом.

Поскольку приложение создало объект **BitMap**, надо не забыть добавить в него обработчик события **OnDestroy** формы, в который вставить оператор

```
BitMap->Free();
```

Процедуру **Timer1Timer** измените следующим образом:

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    Image1->Canvas->Draw(Xpos-L-Rhead,
                        Ypos-L-Hmen-2*Rhead-5, BitMap);
    if ((Xpos >= Image1->Picture->Width - H) | (Xpos <= H))
        revers = -revers;
    Xpos = Xpos + revers * H;
    num = 1 - num;
    BitMap->Canvas->CopyRect(Rect(0,0, BitMap->Width,
                                BitMap->Height), Image1->Canvas,
        Rect(Xpos-L-Rhead,
            Ypos-(L+Hmen+2*Rhead+5),
            Xpos+L+Rhead, Ypos+1));
    Draw();
}
```

Если вы сравните с тем, что было в предыдущем приложении, то увидите, что вместо первого вызова процедуры **Draw**, который стирал предыдущий кадр, вводится оператор **Image1->Canvas->Draw**, который выполняет ту же функцию, но путем восстановления запомненного ранее фрагмента фона под рисунком. Вторым отличием является наличие оператора **BitMap->Canvas->CopyRect**, который перед вызовом **Draw** запоминает новый фрагмент фона.

Вот и все изменения. Выполните проект. Вы увидите, что без фона приложение работает как и прежде. Добавьте пестрый фон так же, как делали это раньше. Вы сможете увидеть, что изображение стало несколько лучше, но не на много. Так

что, как говорилось выше, все равно использовать для мультипликаций пестрые фоны крайне нежелательно.

Наше изображение было очень простым и рисовалось быстро. Но при сложных изображениях время рисования может быть заметным и приводить к мерцанию картинки и другим неприятным зрительным эффектам. В этих случаях используют буферизацию изображения. Это напоминает то, что вы только что делали с фоном, но относится не к фону, а к рисунку. Рисование очередного кадра производится не сразу на холсте, который видит пользователь, а на канве невидимого компонента, типа того **Bitmap**, с которым вы только что работали. А после того, как рисунок сделан, он переносится на видимый холст методами **Draw** или **CopyRect**. Эти методы работают очень быстро и никаких неприятных мерцаний не происходит.

Еще одна проблема анимации — определение того, какие элементы изображаемого объекта видны, а какие — нет. Несмотря на простоту нашего примера с человечком, даже в нем возникла такая проблема, но мы ею пренебрегли, чтобы не усложнять код. Если вы внимательно посмотрите на рис. 5.16 или 5.17, то увидите, что изображение неправильное. Конец одной из рук должен быть скрыт за лаврами, которые держит человечек. Эту ошибку в данном случае не трудно было бы убрать, но код несколько усложнился бы. А вот в трехмерной графике при вращении изображения объекта подобная проблема встает очень остро и должна соответствующим образом решаться.

Последний вопрос, который мы рассмотрим, — как сделать нашу мультипликацию более плавной. Если вы не стремитесь к лаврам Диснея, вам достаточно ограничиться в ваших мультипликациях простыми механическими движениями, которые всегда можно описать соответствующими функциями. Это относится к любым динамическим иллюстрациям работы механизмов, к любым схематическим перемещениям. Можно применить функциональное описание движения и к нашему человечку. Вполне допустимо считать, что его руки и ноги движутся по окружностям с соответствующими центрами. Тогда легко рассчитать их положение в любой момент времени и соответственно разбить это движение на любое число кадров.

Давайте сделаем движения нашего человечка более плавными. Откройте опять ваше первое приложение с анимацией и опять сохраните модуль и сам проект под новыми именами. Вы можете ничего не добавлять на форму, а только изменить тексты процедур. Теперь эти тексты должны иметь следующий вид.

```
#define Pi 3.1415926535897932385
short int cadr = 0;           // номер кадра
short int H=30;               // длина ноги и руки
short int Xpos = 3 * H;       // координата опорной ноги
short int Ypos = 120;         // «земля»
short int Hmen = 30;          // высота тела
short int Rhead = 10;         // радиус головы
short int revers = 1;          // направление движения
short int L = H * 1.41;       // длина ноги
short int Ncadr=16;           // число кадров на шаг
//-----
void __fastcall TForm1::Draw()
{
    float Angl = Pi/4*(1+(2.*cadr)/(Ncadr-1));
    short int Yb = Ypos-H*sin(Angl);
    short int Yt = Yb-Hmen;
    short int X = Xpos-revers*H*cos(Angl);
    Image1->Canvas->MoveTo(X-(Xpos-X),Ypos);
    Image1->Canvas->LineTo(X,Yb);           // нога

    if (cadr != Ncadr / 2-1)
        Image1->Canvas->LineTo(Xpos,Ypos); // другая нога
    Image1->Canvas->MoveTo(X,Yb);
```

```

Imagel->Canvas->LineTo(X,Yt); // туловище
short int Xl = X - revers * (Yb-Ypos);
Imagel->Canvas->MoveTo(Xl,Yt+5-(Xpos-X));
Imagel->Canvas->Ellipse(Xl-Rhead / 2,
                      Yt+5-(Xpos-X)-Rhead / 2,
                      Xl+Rhead / 2,
                      Yt+5-(Xpos-X)+Rhead / 2);
Imagel->Canvas->LineTo(X,Yt+5); // рука

if (cadr != Ncadr / 2-1)
{
    Imagel->Canvas->Ellipse(Xl-Rhead / 2,
                          Yt+5+(Xpos-X)-Rhead / 2,
                          Xl+Rhead / 2,
                          Yt+5+(Xpos-X)+Rhead / 2);
    Imagel->Canvas->LineTo(Xl,Yt+5+(Xpos-X)); // другая рука
}
Imagel->Canvas->Ellipse(X-Rhead,Yt-2*Rhead,X+Rhead,Yt);
Imagel->Canvas->Rectangle(X-Rhead,Yt-2*Rhead-4,
                        X+Rhead,Yt-2*Rhead-1); // шляпа
}
//-----
void __fastcall TForm1::BRunClick(TObject *Sender)
{
    Timer1->Enabled = ! Timer1->Enabled;
}
//-----
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    Draw();
    cadr = (cadr+1) % Ncadr;
    if (cadr == 0)
        if ((Xpos < Imagel->Picture->Width-revers*3*H) &&
            (Xpos > -revers*3*H))
            Xpos += revers*H*1.41;
        else revers = -revers;
    Draw();
}
//-----
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Imagel->Canvas->MoveTo(0,Ypos+3);
    Imagel->Canvas->Pen->Width = 4;
    Imagel->Canvas->LineTo(Imagel->ClientWidth,Ypos+3); // земля
    Imagel->Canvas->Pen->Width = 1;
    Imagel->Canvas->Pen->Mode = pmNotXor;
    Timer1->Interval = 600 / Ncadr;
    Draw();
}

```

В этом коде использованы математические функции **sin** и **cos**. Поэтому в код надо добавить директиву препроцессора

```
#include <math.h>
```

Рассмотрим коротко приведенный код.

Процедура **FormCreate** отличается от той, что была в первом приложении, только оператором, задающим выдержку таймера (свойство **Interval**) путем деления 600 на константу **Ncadr**, которая задает число кадров на один цикл движения — на один шаг человечка. Как видно, длительность одного шага выбрана равной 600 миллисекунд. Вы, конечно, можете изменить это значение, как и значение **Ncadr**, выбранное равным 16.

Процедура **BRunClick** не отличается от той, что была в первом приложении. Процедура **Timer1Timer**, как и в первом приложении начинается и кончается вызовами **Draw**, первый из которых стирает изображение предыдущего кадра, а второй — рисует новый кадр. После первого вызова **Draw** рассчитывается значение переменной **cadr** оператором

```
cadr = (cadr+1) % Ncadr;
```

Поскольку тут используется операция вычисления остатка от деления **cadr+1** на **Ncadr**, то значение **cadr** последовательно получает значения 0, 1, 2, ..., **Ncadr** — 1, 0, 1, Шаг начинается с **cadr** = 0. При этом проверяется, не приблизился ли человек к краю формы, и если приблизился — изменяется направление движения (знак переменной **revers**).

Наиболее серьезно изменилась процедура **Draw**. Она начинается с определения угла наклона ног и рук **Angl**, исходя из номера кадра:

```
float Angl = Pi/4*(1+(2.*cadr)/(Ncadr-1));
```

Обратите внимание на то, что константа 2 в этом выражении записана с точкой и произведение **2.*cadr** заключено в скобки. Это принципиально, так как в этом случае константа воспринимается как значение с плавающей запятой и все выражение **(2.*cadr)/(Ncadr-1)** вычисляется как значение с плавающей запятой. Если не поставить точку после 2, то будут применяться целочисленные вычисления и пока **2*cadr** меньше, чем **Ncadr-1**, значения этого выражения будут равны 0 и угол изменяться не будет.

После вычисления угла на его основе строится изображение, подобное тому, которое было в первом приложении. Обратите внимание на то, что вторая нога и вторая рука человечка рисуются только, если выполняется условие **cadr != Ncadr/2-1**. Это связано с тем, что, если число кадров **Ncadr** четное, то в этот момент одна нога накладывается на другую и руки также накладываются друг на друга. Поскольку рисование идет в режиме **pmNotXor**, то это наложение приведет к тому, что у человечка вообще исчезнут в этом кадре руки и ноги. Правда, всего на один кадр, но все равно неприятно.

Ваше приложение готово. Можете сохранить его и выполнить. Вы увидите, что движения человечка стали плавными.

5.2.3 Воспроизведение немых видео клипов — компонент **Animate**

Теперь рассмотрим способ воспроизведения в приложении C++Builder стандартных мультимедийных Windows и файлов **.avi** — клипов без звукового сопровождения. Это позволяет сделать компонент **Animate**, расположенный на странице Win32 библиотеки.

Компонент **Animate** позволяет воспроизводить на форме стандартные видео клипы Windows (типа копирования файлов, поиска файлов и т.п.) и немые видео файлы **.avi** — Audio Video Interleaved. Эти файлы представляют собой последовательность кадров битовых матриц. Они могут содержать и звуковую дорожку, но компонент **Animate** воспроизводит только немые клипы AVI. Он работает только с неуплотненными файлами AVI или с клипами AVI, уплотненными с использованием RLE — run-length encoding.

TAnimate может воспроизводить клипы AVI из ресурсов, из файлов или из библиотеки **Shell32.dll**, если приложение работает с Windows 95 или NT.

Откройте новое приложение, перенесите на форму компонент **Animate** и познакомьтесь с ним.

Воспроизводимое им изображение задается одним из двух свойств: **FileName** или **CommonAVI**. Первое из этих свойств, как ясно из его названия, позволяет в процессе проектирования или программно задать имя воспроизводимого файла. А

свойство **CommonAVI** позволяет воспроизводить стандартные мультипликации Windows. Это свойство типа **TCommonAVI**, объявленного следующим образом:

```
enum TCommonAVI {aviNone, aviFindFolder, aviFindFile,
                 aviFindComputer, aviCopyFiles, aviCopyFile,
                 aviRecycleFile, aviEmptyRecycle,
                 aviDeleteFile };
```

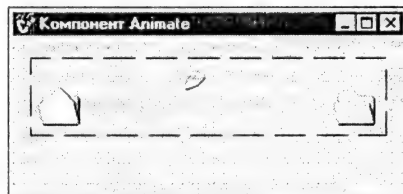
Тип **TCommonAVI** определяет множество предопределенных в Windows мультипликаций типа копирования файлов, поиска файлов, удаления файлов и т.п. Что означает каждое значение вы увидите из тестового приложения, которое построите чуть позже.

А пока установите значение **CommonAVI**, например, равным **aviCopyFile**. Это соответствует стандартному изображению копирования файла. Соответствующий начальный рисунок немедленно появится на вашей форме. Свойство **Repetitions** компонента **Animate** задает число повторений воспроизведения клипа. Если оно равно 0 (значение по умолчанию), то воспроизведение повторяется вновь и вновь до тех пор, пока не будет выполнен метод **Stop**. При выполнении этого метода генерируется событие **OnStop**, которое можно использовать, например, чтобы стереть изображение — сделать его невидимым.

Если же свойство **Repetitions** задать большим нуля, оно определит число повторений клипа. Задайте его, например, равным 3. А теперь установите свойство **Active** компонента **Animate** в **true**. Вы увидите (рис. 5.18), что еще в процессе проектирования ваше приложение заработает. Изображение оживет и клип будет повторен 3 раза.

Рис. 5.18

Анимация копирования файла в процессе проектирования



Вы можете посмотреть воспроизводимое изображение по кадрам. Для этого щелкните на компоненте правой кнопкой мыши и из всплывшего меню выберите разделы **Next Frame** (следующий кадр) или **Previous Frame** (предыдущий кадр). Это позволит вам выбрать фрагмент клипа, если вы не хотите воспроизводить клип полностью. Воспроизвести фрагмент клипа можно, установив соответствующие значения свойств **StartFrame** — начальный кадр воспроизведения, и **StopFrame** — последний кадр воспроизведения.

Воспроизводить фрагмент клипа можно и методом **Play**, который определен следующим образом:

```
void __fastcall Play(Word FromFrame, Word ToFrame, int Count);
```

Метод воспроизводит заданную последовательность кадров клипа от **FromFrame** до **ToFrame** включительно и воспроизведение повторяется **Count** раз. Если **FromFrame** = 1, то воспроизведение начинается с первого кадра. Значение **ToFrame** должно быть не меньше **FromFrame** и не больше значения, определяемого свойством **FrameCount** (свойство только для чтения), указывающим полное число кадров в клипе. Если **Count** = 0, то воспроизведение повторяется до тех пор, пока не будет выполнен метод **Stop**.

Выполнение **Play** идентично заданию **StartFrame** равным **FromFrame**, **StopFrame** равным **ToFrame**, **Repetitions** равным **Count** и последующей установке **Active** в **true**.

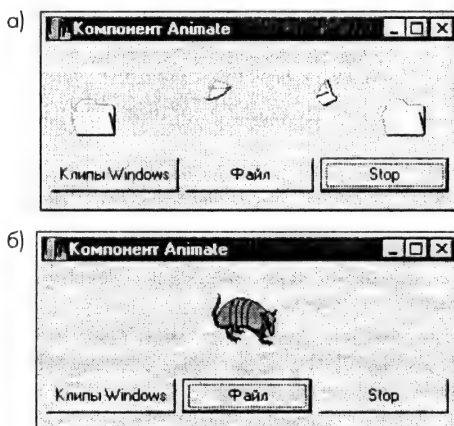
В компоненте **Animate** предусмотрены события **OnClose**, **OnOpen**, **OnStart** и **OnStop**, генерируемые соответственно в моменты закрытия и открытия компонента, начала и окончания воспроизведения.

Давайте теперь построим тестовое приложение, показывающее возможности компонента **Animate**. Установите в том приложении, которое вы уже начали, свойство **Visible** компонента **Animate** в **false**. Это надо для того, чтобы изображение возникало только тогда, когда произойдет соответствующее событие: копирование файлов, поиск файлов и т.п. В тестовом приложении мы будем имитировать начало и окончание события, которое должно сопровождаться мультипликацией, нажатиями кнопок запуска и останова воспроизведения. Поэтому верните значение свойства **Repetitions** в 0, чтобы воспроизведение длилось до окончания события. Свойство **Active** установите в **false**. Полезно также установить свойство **AutoSize** в **false**, а свойство **Center** в **true**, чтобы изображение всегда появлялось в центре экрана.

А теперь добавьте в приложение 3 кнопки (рис. 5.19). Первая из них (назовите ее **BWind**) будет начинать процесс воспроизведения поочередно всех стандартных клипов Windows. Вторая кнопка (назовите ее **BStop**) пусть завершает воспроизведение очередного клипа. А третью кнопку (назовите ее **BFile**) введем для того, чтобы показать, что компонент может воспроизводить изображения из заданного файла **.avi**. Чтобы пользователь мог выбрать файл изображения, добавьте на форму компонент **OpenDialog** и задайте его фильтр (свойство **Filter**) равным «видео *.avi*.avi».

Рис. 5.19

Демонстрация возможностей компонента **Animate**: воспроизведение стандартного клипа Windows (а) и воспроизведение файла **.avi** (б)



Теперь все приготовления закончены и осталось только написать обработчики событий. Код обработчиков может иметь вид:

```
int i;
//-----
void __fastcall TForm1::BWindClick(TObject *Sender)
{
    Animate1->Visible = true;
    i = 1;
    Animate1->CommonAVI = aviFindFolder;
    Animate1->Active = true;
}
//-----
void __fastcall TForm1::BStopClick(TObject *Sender)
{
    Animate1->Stop();
}
//-----
```

```

void __fastcall TForm1::Animate1Stop(TObject *Sender)
{
    i++;
    switch (i)
    {
        case 2: Animate1->CommonAVI = aviFindFile;
                break;
        case 3: Animate1->CommonAVI = aviFindComputer;
                break;
        case 4: Animate1->CommonAVI = aviCopyFiles;
                break;
        case 5: Animate1->CommonAVI = aviCopyFile;
                break;
        case 6: Animate1->CommonAVI = aviRecycleFile;
                break;
        case 7: Animate1->CommonAVI = aviEmptyRecycle;
                break;
        case 8: Animate1->CommonAVI = aviDeleteFile;
    }
    if (i < 9) Animate1->Active = true;
    else Animate1->Visible = false;
}
//-----
void __fastcall TForm1::BFileClick(TObject *Sender)
{
    if (OpenDialog1->Execute())
    {
        i = 9;
        Animate1->FileName = OpenDialog1->FileName;
        Animate1->Visible = true;
        Animate1->Active = true;
    }
}

```

Обработчик события **OnClick** кнопки **BWind** задает начальное значение свойства **CommonAVI**, сбрасывает счетчик на 1, делает компонент **Animate1** видимым и активизирует его.

Обработчик события **OnClick** кнопки **BStop** останавливает воспроизведение методом **Stop**.

Обработчик события **OnStop** компонента **Animate1** увеличивает счетчик на 1, в зависимости от значения счетчика загружает в компонент соответствующий клип Windows, и активизирует компонент. Если все клипы уже воспроизведены, то компонент делается невидимым.

Обработчик события **OnClick** кнопки **BFile** загружает в компонент видеофайл, выбранный пользователем.

Выполните приложение и проверьте его в работе. В качестве видео файла можете использовать файл...\\Examples\\Mfc\\General\\Cmnctrls\\dillo.avi, поставляемый с примерами C++Builder (на рис. 5.19 б изображен момент воспроизведения именно этого файла), или любой другой более интересный видеофайл.

5.2.4 Универсальный проигрыватель MediaPlayer

В C++Builder имеется компонент **MediaPlayer** — универсальный проигрыватель аудио- и видео- информации. Этот медиа-плеер расположен на странице System библиотеки компонентов. Он инкапсулирует интерфейс управления носителями (Media Control Interface — MCI) Windows 95 и Windows NT.

Компонент можно использовать в двух режимах. Во-первых, можно предоставить пользователю возможность управлять воспроизведением информации с помощью кнопочного интерфейса, напоминающего панель управления различными

проигрывателями. Во-вторых, можно сделать сам компонент невидимым и управлять воспроизведением информации с помощью его методов.

Пользовательский интерфейс медиа-плеера представлен на рис. 5.20. Он имеет ряд кнопок, управляемых мышью или клавишей пробела и клавишами со стрелками.

Рис. 5.20
Панель компонента MediaPlayer



Назначение кнопок (перечисляются слева направо):

Кнопка	Действие
Play	Воспроизведение
Pause	Пауза воспроизведения или записи. Если медиа-плеер в момент щелчка уже в состоянии паузы, то воспроизведение или запись возобновляются
Stop	Останов воспроизведения или записи
Next	Переход на следующий трек или на конец
Prev	Переход на предыдущий трек или на начало
Step	Перемещение вперед на заданное число кадров
Back	Перемещение назад на заданное число кадров
Record	Начало записи
Eject	Освобождение объекта, загруженного в устройство

Каждой кнопке медиа-плеера соответствует метод, осуществляющий по умолчанию требуемую операцию: **Play**, **Pause**, **Stop**, **Next**, **Previous**, **Step**, **Back**, **StartRecording**, **Eject**.

Тип устройства мультимедиа, с которым работает медиа-плеер, определяется его свойством **DeviceType**. Если устройство мультимедиа хранит объект воспроизведения в файле, то имя файла задается свойством **FileName**. По умолчанию свойство **DeviceType** имеет значение **dtAutoSelect**. Это означает, что медиа-плеер пытается определить тип устройства исходя из расширения имени файла **FileName**.

Еще одно свойство **MediaPlayer** — **AutoOpen**. Если оно установлено в **true**, то медиа-плеер пытается открыть устройство, указанное свойством **DeviceType**, автоматически во время своего создания в процессе выполнения приложения.

Воспроизведение видео информации по умолчанию производится в окно, которое создает само открытое устройство мультимедиа (см. приведенный далее рис. 5.21 б). Однако это можно изменить, если в свойстве **Display** указать оконный элемент, в котором должно отображаться изображение. Это может быть, например, форма или панель. Можно также задать в свойстве **DisplayRect** типа **TRect** (свойство только времени выполнения) прямоугольную область этого окна, в которую должно выводиться изображение. Для задания свойства **DisplayRect** можно использовать функцию **Rect**. Однако, в данном свойстве использование этого типа не совсем обычно. Первые две координаты, как и обычно, задают положение левого верхнего угла изображения. А два следующих числа задают ширину и высоту изображения, а не координаты правого нижнего угла. Например, оператор

```
MediaPlayer1->DisplayRect = Rect(10,10,200,200);
```

задает для вывода область с координатами левого верхнего угла (10, 10), длиной и шириной, равными 200.

В компоненте **MediaPlayer** определены события **OnClick** и **OnNotify**. Первое из них происходит при выборе пользователем одной из кнопок медиа-плеера и определено как

```
enum TMPBtnType {btPlay, btPause, btStop, btNext,
                 btPrev, btStep, btBack, btRecord, btEject};

void __fastcall Click(TObject *Sender, TMPBtnType Button,
                    bool &DoDefault)
```

Параметр **Button** указывает выбранную кнопку. Параметр **DoDefault**, передаваемый по ссылке, определяет выполнение (при значении **true** по умолчанию) или отказ от выполнения стандартного метода, соответствующего выбранной кнопке.

Событие **OnNotify** происходит после возвращения очередного метода, если свойство медиа-плеера **Notify** было установлено в **true**. Способ возврата любого метода медиа-плеера определяется свойством **Wait**. Если установить **Wait** равным **false**, то возвращение управления в приложение происходит сразу после вызова метода, не дожидаясь завершения его выполнения. Таким образом, задав **Notify** равным **true** и **Wait** равным **false**, можно обеспечить немедленный возврат в приложение и отображения пользователю текущего состояния объекта мультимедиа.

Свойства **Notify** и **Wait** действуют только на один очередной метод. Поэтому их значения надо каждый раз восстанавливать в обработчиках событий **OnClick** или **OnNotify**.

В обработчиках событий можно читать свойство **Mode**, характеризующее текущее состояние устройства мультимедиа. Можно также читать и устанавливать ряд свойств, характеризующих размер воспроизводимого файла и текущую позицию в нем.

Вот, собственно, в конспективном виде основная информация о компоненте **MediaPlayer**. А теперь попробуйте все это на практике. Простое и в то же время мощное приложение можно сделать очень просто. Начните новый проект и перенесите на форму компоненты **MediaPlayer**, **MainMenu** и **OpenDialog**. В фильтре компонента **OpenDialog** можно, например, задать:

аудио и видео (*.wav,*.mid,*.avi)	*.wav; *.mid; *.avi
аудио (*.wav,*.mid)	*.wav;*.mid
видео (*.avi)	*.avi
все файлы	*.*

В меню достаточно задать одну команду: Файл | Открыть. Обработчик события **OnClick** этой команды может содержать оператор

```
if (OpenDialog1->Execute())
{
    MediaPlayer1->FileName = OpenDialog1->FileName;
    MediaPlayer1->Open();
}
```

который открывает устройство мультимедиа, соответствующее выбранному пользователем файлу. При этом надо проследить, чтобы в компоненте **MediaPlayer** свойство **DeviceType** равнялось **dtAutoSelect**. Это обеспечит автоматический выбор соответствующего устройства мультимедиа исходя из расширения выбранного файла.

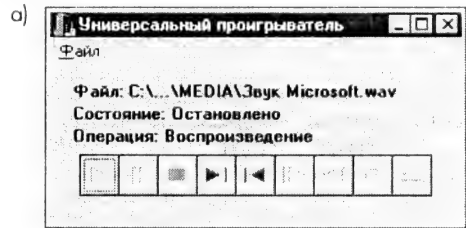
В компоненте **MediaPlayer** при желании можно указать имя файла **FileName**, открываемого в момент начала выполнения приложения. Тогда надо установить свойство **AutoOpen** в **true**. Впрочем, это, конечно, не обязательно.

Вот и все. Можете выполнять свое приложение и наслаждаться музыкой или фильмами (если, конечно, все вопросы, связанные с настройкой мультимедиа на вашем компьютере решены).

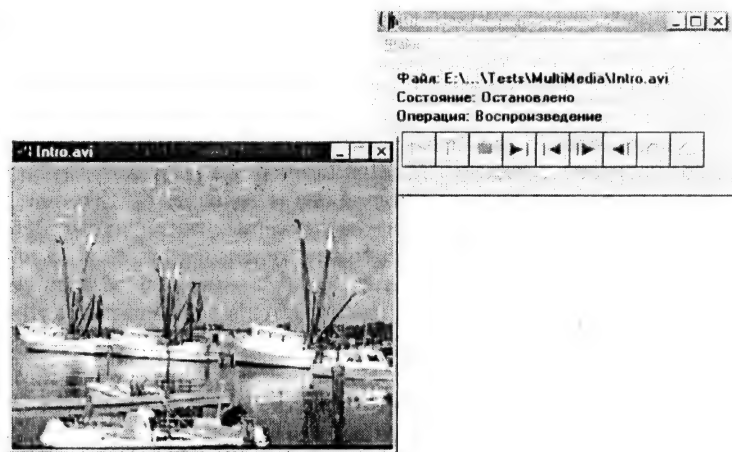
Чтобы все-таки использовать какие-то события компонента **MediaPlayer**, давайте немного усложним приложение. Введем в него четыре метки (рис. 5.21). В первой из них (**Label1**) укажем надпись «Файл:». Во второй (**Label2**) будем программно отображать состояние проигрывателя, в третьей (**Label3**) — последнюю вызванную операцию. Четвертую метку (**Label4**) расположим рядом с меткой **Label1** так, чтобы она служила ее продолжением. В ней мы будем отображать имя загруженного файла, но в сокращенном виде с многоточиями (см. рис. 5.21), если имя файла не помещается в отведенном для него месте.

Рис. 5.21

Приложение универсального проигрывателя: воспроизведение аудио файла (а) и видео файла в отдельном окне (б)



б)



Код, обеспечивающий подобную обратную связь в приложении, может быть следующим.

```
#include «filectrl.hpp»
AnsiString ModeStr[7] =
{
    "Не готово", "Остановлено", "Воспроизведение",
    "Запись", "Поиск", "Пауза", "Открыто"};
AnsiString ButtonStr[9] =
{
    "Воспроизведение", "Пауза", "Стоп",
    "Следующий", "Предыдущий", "Вперед", "Назад",
    "Запись", "Конец"};
//_____
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Label4->Caption = MinimizeName(MediaPlayer1->FileName,
                                   Label4->Canvas, 200);
    Label2->Caption = "Состояние: " +
                     ModeStr[MediaPlayer1->Mode];
    MediaPlayer1->Notify = true;
```

```

}
//-----
void __fastcall TForm1::OpenClick(TObject *Sender)
{
    if (OpenDialog1->Execute())
    {
        MediaPlayer1->FileName = OpenDialog1->FileName;
        Label4->Caption = MinimizeName(MediaPlayer1->FileName,
                                     Label4->Canvas, 200);
        MediaPlayer1->Open();
        MediaPlayer1->Notify = true;
    }
}
//-----
void __fastcall TForm1::MediaPlayer1Notify(TObject *Sender)
{
    Label2->Caption = "Состояние: " + ModeStr[MediaPlayer1->Mode];
    // Переустановка Notify,
    // чтобы событие произошло в следующий раз
    MediaPlayer1->Notify = true;
}
//-----
void __fastcall TForm1::MediaPlayer1Click(TObject *Sender,
                                           TMouseButton Button, bool &DoDefault)
{
    Label3->Caption = "Операция: " + ButtonStr[Button];
    // Переустановка Notify, чтобы произошло событие OnNotify
    MediaPlayer1->Notify = true;
}

```

В свойстве **FileName** компонента **MediaPlayer1** задано имя файла, загружаемого в момент создания **MediaPlayer1**, т.е. в момент создания приложения. Соответственно в обработчике **FormCreate** события **OnCreate** формы записаны операторы, задающие имя файла и текущее состояние проигрывателя в метки **Label4** и **Label2** соответственно. Для записи имени файла использована функция **MinimizeName**, которая обеспечивает сокращенное отображение пути к файлу (см. рис. 5.21) в случае, если полный путь не помещается в отведенном месте (в операторе указана максимальная длина — 200 пикселей). Чтобы эта функция работала, в модуль добавлена директива компилятора

```
#include "filectrl.hpp"
```

В дальнейшем отображение соответствующей информации предусмотрено в процедурах, соответствующих открытию файла (**OpenClick**), нажатию пользователем какой-нибудь кнопки проигрывателя (**MediaPlayer1Click**), возвращении любого метода проигрывателя (**MediaPlayer1Notify**). После каждого события выполняется оператор

```
MediaPlayer1->Notify = true;
```

обеспечивающий наличие события **OnNotify** после возвращения следующего метода проигрывателя.

Запустите приложение и проверьте его в работе (рис. 5.21). С его помощью вы можете слушать музыку, смотреть немые и звуковые клипы, т.е. можете наслаждаться всеми прелестями мультимедиа.

Глава 6

Взаимодействие приложения с внешними программами

6.1 Запуск из приложения внешних программ

Взаимодействие приложения с внешними программами может происходить различными способами. Это может быть:

- непосредственный запуск внешней программы из вашего приложения
- запуск внешней программы, связанной с некоторым документом
- обмен с приложениями сообщениями Windows
- технология OLE — внедрения и связывания документов, подготовленных внешними программами, в ваше приложение
- использование новых компонентов C++Builder 5 — серверов COM
- динамический обмен данными между приложениями — DDE

Начнем с рассмотрения первой из этих возможностей — запуска из приложения внешней программы. Такая необходимость может возникать в ряде случаев. Например, ваша задача может заключаться в создании с помощью C++Builder пользовательского интерфейса к какой-нибудь ранее разработанной для MS-DOS или Windows программе, которая представлена только своим выполняемым модулем. Исходный текст программы может быть при этом вам не известен или он может быть написан на языке, отличающемся от C и C++. Тогда естественным способом решения этой задачи является запуск загрузочного модуля программы из вашего приложения — интерфейса. Другой пример использования внешней программы — желание предоставить пользователю, работающему с вашей программой, какие-то дополнительные возможности. Например, вам может захотеться дать пользователю возможность вызвать из вашего приложения программу Windows «Калькулятор», чтобы просчитать какие-то данные, которые он получил в процессе работы с приложением, и принять решение о своих дальнейших действиях.

Запуск внешней программы из приложения C++Builder можно сделать несколькими способами. Ниже рассматриваются наиболее простые из них.

6.1.1 Запуск внешней программы функцией `execvp`

Функция `execvp` позволяет выполнить из своего приложения любое указанное приложение, передав ему управление. Вызванная программа замещает в памяти вызвавшее ее приложение. Таким образом, родительское приложение завершается и начинается новое.

Функция `execvp` определена в файле `process.h` (не забывайте включать его в приложение соответствующей директивой `#include`) следующим образом:

```
int execvp(char *path, char *arg0, *arg1, ..., *argn, NULL)
```

Параметр `path` определяет имя и путь к приложению, которое требуется выполнить. Если в `path` указан путь и имя файла с расширением, то функция ищет именно этот файл. Если же расширение файла не задано, то сначала ищется файл такой, который задан. Если он не находится, к имени добавляется расширение

.exe и поиск повторяется. Если файл опять не находится, к имени добавляется расширение .com и поиск повторяется. Если в **path** не задан путь, то сначала поиск файла производится в текущем каталоге. Если в нем требуемый файл не найден, то поиск продолжается в каталогах, указанных в переменной окружения **PATH**.

Аргументы функции **arg0** — **argn** являются параметрами, передаваемыми в запускаемую на выполнение программу через командную строку. Функция должна передать в запускаемое приложение хотя бы один аргумент **arg0**. По соглашению этот аргумент — копия **path**. Впрочем, передача другого значения не является ошибкой. Остальные аргументы, если они требуются, передают в запускаемую программу дополнительную информацию. Если она состоит из нескольких слов, например, из нескольких опций, то можете каждое слово передавать отдельным аргументом, а может все их объединить в одну строку **arg1**. Последний аргумент функции **exec1p** — **NULL** является признаком окончания списка аргументов.

Функция **exec1p** возвращает 0 при успешной загрузке нового приложения, а при ошибке возвращает -1.

Рассмотрим примеры использования функции **exec1p**. Оператор

```
if (exec1p("F1.exe", "F1.exe", NULL))  
    ShowMessage("Программа F1.exe не выполнена");
```

завершает выполнение вашего приложения и передает управление программе с выполняемым файлом **F1.exe**. Этот файл должен быть расположен в рабочем каталоге или в одном из каталогов, указанных в переменной окружения **PATH**. Иначе функция **exec1p** вернет -1 и будет выдано сообщение функцией **ShowMessage**. Аналогичное сообщение будет выдано если, например, для загрузки **F1.exe** не хватает оперативной памяти.

Оператор

```
exec1p("nc", "nc", NULL);
```

передает управление программе Norton Commander (файл **nc.exe**), если только путь к этой программе указан в переменной окружения **PATH**.

Оператор

```
char * prog = "command.com";  
exec1p(prog, prog, NULL);
```

передает управление DOS, если только путь к файлу **command.com** указан в переменной окружения **PATH**.

Оператор

```
exec1p("Winword", "Winword", "F1.doc", "F2.doc", NULL))
```

запускает редактор Word и передает в него файлы **F1.doc** и **F2.doc**.

Мы рассмотрели только одну функцию **exec1p** — из целого семейства, включающего 8 подобных функций. Подробную информацию о них вы найдете в главе 15 в разделе 15.6.2. По своим основным характеристикам все эти функции аналогичны **exec1p**, различаясь только каталогами, в которых ищется файл, передачей в новое приложение переменных окружения и заданием аргументов функции.

Рассмотренные функции могут найти достаточно ограниченное применение, поскольку они обеспечивают безвозвратную передачу управления из вызвавшего приложение в новое. И для возврата в исходное приложение надо принимать специальные меры: например, вызванное приложение в конце своей работы должно аналогичной функцией **exec1p** вызвать первоначальное приложение. Зато у этих функций есть и большое преимущество — оверлейная загрузка приложений. Новое приложение загружается в оперативную память на место вызвавшего его приложения. Соответственно сокращаются затраты памяти, так как не требуется держать в ней оба приложения.

Таким образом, сфера применения функции **exec1p**:

- построение входного интерфейса к какому-то приложению, работающего только перед запуском этого приложения
- создание оверлейных приложений, загружаемых в память по частям

6.1.2 Запуск внешней программы функцией `spawnlp`

Функция `spawnlp` подобна рассмотренной выше функции `execsp`, но обладает более широкими возможностями. Она и используемые в ней константы описаны в файлах `process.h` и `stdio.h`. Объявление функции:

```
int spawnlp(int mode, char *path, char *arg0, arg1, ..., argn, NULL)
```

Функция `spawnlp` отличается от `execsp` наличием параметра `mode`, задающего режим выполнения приложения, запускаемого на выполнение. Этот параметр может, в частности, принимать следующие значения (полный их список см. в главе 15 в разделе 15.6.2):

P_WAIT	Родительское приложение ждет завершения вызванного приложения, после чего продолжается его выполнение
P_NOWAIT	Родительское приложение продолжает выполняться пока выполняется вызванное приложение. Этот режим недоступен в 16-разрядных Windows и DOS
P_DETACH	Идентичен P_NOWAIT , но вызванное приложение выполняется в фоновом режиме, так что не имеет доступа к клавиатуре и дисплею
P_OVERLAY	Вызванное приложение замещает в памяти родительское. То же, что вызов функции <code>execsp</code>

Приведем пример. Операторы

```
if(spawnlp(P_WAIT,"arj","arj","e doc.arj a1.txt", NULL))
    ShowMessage("Программа arj не выполнена");
else
{
    Memol->Clear();
    Memol->Lines->LoadFromFile("a1.txt");
    DeleteFile("a1.txt");
}
```

запускают архиватор `arj`, извлекающий из архива `doc.arj` файл `a1.txt`. Приложение ждет, пока программа `arj` закончит работу, затем загружает разархивированный файл в окно редактирования `Memol` и удаляет этот файл с диска.

В приведенном примере все аргументы, передаваемые в порождаемый процесс, объединены в одной строке. Тот же самый результат получился бы, если передать их все в отдельности:

```
if(spawnlp(P_WAIT,"arj","arj","e","doc.arj","a1.txt", NULL))
    ...
```

Операции, подобные рассмотренным выше, невозможно было бы выполнить функцией `execsp`, поскольку она не обеспечивает возвращение в исходное приложение. Нельзя было бы выполнить эти операции и функцией `spawnlp` при режиме, отличном от **P_WAIT**, поскольку в этом случае оператор загрузки файла в окно редактирования выполнялся бы раньше, чем успевал распаковываться архив.

Надо отметить, что приведенный выше пример разархивации файла обладает двумя недостатками. Первый из них связан с тем, что выполняется программа `arj`, предназначенная для DOS. Поэтому при ее выполнении вызывается сеанс DOS, и

после его окончания пользователь видит окно DOS, которое ему надо закрыть, чтобы продолжить работу. Это, конечно, очень неудобно. Устранить этот недостаток легко, например, написанием пакетного файла **arj.bat** вида:

```
@echo off
arj.exe e doc %1
exit
```

В нем помимо команда разархивации предусмотрена команда **exit** — окончание сеанса работы с окном DOS. Тогда обращение к разархивации в приложении может быть даже короче, чем раньше:

```
if (spawnlp(P_WAIT, "arj.bat", "arj.bat", "al.txt", NULL))
...
```

Обращение к пакетному файлу **arj.bat** позволяет порожденному процессу автоматически, без вмешательства пользователя вернуться в родительский процесс. Но остается еще один недостаток рассмотренного примера — на время выполнения разархивации получаются неприятные изменения экрана, связанные с выходом в DOS. Этот недостаток может быть снят только функциями, рассмотренными в следующих разделах.

Приведем еще один пример использования функции **spawnlp**. Пусть вы разработали пользовательский интерфейс, в котором хотите предоставить пользователю возможность запускать различные приложения. Ваш интерфейс достаточно большой и поэтому желательно запускать из него внешние приложения в оверлэйном режиме. Эту задачу можно решить следующим образом.

Пусть имя вашего приложения **POverlay**. Создайте еще одно приложение, названное, например, **OMenage**. Это приложение будет управлять запуском требуемых программ. Оно может быть очень маленьким, не содержать ни одной формы и располагаться в оперативной памяти одновременно с запускаемыми программами. Весь текст его файла следующий:

```
#include <vcl.h>
#pragma hdrstop
#include <process.h>
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR lpCmdLine, int)
{
    spawnlp(P_WAIT, lpCmdLine, lpCmdLine, NULL);
    spawnlp(P_OVERLAY, "POverlay.exe", "POverlay.exe", NULL);
    return 0;
}
```

Первый вызов функции **spawnlp** обеспечивает запуск в режиме ожидания того приложения, имя которого передано через командную строку **lpCmdLine**. Вторым вызов **spawnlp** обеспечивает оверлэйный вызов вашего основного приложения **POverlay.exe**.

Предположим, что в вашем основном приложении **POverlay** имя запускаемой программы записано в окне редактирования **Edit1**. Тогда вызов этой программы может осуществляться оператором:

```
if (spawnlp(P_OVERLAY, "OMenage.exe", "OMenage.exe", Edit1->Text,
            NULL))
    ShowMessage("Программа " + Edit1->Text + " не выполнена;" +
               " нет файла OMenage.exe");
```

Этот оператор прервет выполнение приложения **POverlay** и загрузит на его место в памяти короткую (примерно 10 К) программу **OMenage.exe**, передав в нее как параметр имя запускаемого приложения. Программа **OMenage.exe** вызовет в режиме ожидания эту программу, а по окончании ее работы удалится из памяти и опять вызовет основное приложение **POverlay**. Таким образом, во время выполнения вызываемой программы в памяти будет находиться не ваше большое приложение **POverlay**, а только маленькая программа управления **OMenage.exe**.

Описанное взаимодействие программ имеет некоторый недостаток: при возврате в **POverlay** текст в окне **Edit1** будет утерян. Этот недостаток легко устранить. Измените основной файл приложения **POverlay** следующим образом:

```
#include <vcl.h>
#pragma hdrstop
USERES("POverlay.res");
USEFORM("UOverlay1.cpp", Form1);
#include "UOverlay1.h" // включение головного файла приложения

//-----
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR lpCmdLine, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TForm1), &Form1);
        Form1->Edit1->Text = lpCmdLine; // Загрузка окна Edit1
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
```

По сравнению со стандартным файлом, созданным C++Builder, в него добавлено две строки (отмечены комментариями): директива, включающая головной файл модуля **UOverlay1.h**, содержащего описание вашей формы **Form1**, и оператор, загружающий в окно **Edit1** текст, переданный через командную строку. Еще одно изменение по сравнению со стандартным файлом — введение в заголовок функции **WinMain** параметра **lpCmdLine** — ссылки на командную строку. Если в файле приложения **POverlay** сделаны такие изменения, то в приложении **OMenage** второй вызов функции должен быть изменен на следующий:

```
spawnlp(P_OVERLAY, "POVERLAY.exe", "POVERLAY.exe", lpCmdLine, NULL);
```

Этот вызов отличается от того, что был раньше, передачей в программу той командной строки, которая была задана при вызове **OMenage**. Таким образом в программу **POverlay** вернется имя запускавшейся программы, которое будет загружено в окно **Edit1**.

Функция **spawnlp** принадлежит к целому семейству, включающему 8 подобных функций. Подробную информацию о них вы найдете в главе 15 в разделе 15.6.2. По своим основным характеристикам все эти функции аналогичны, различаясь только каталогами, в которых ищется файл, передачей в новое приложение переменных окружения и заданием аргументов функции.

6.1.3 Запуск внешней программы функцией WinExec

Функция **WinExec**, в отличие от описанных в предыдущих разделах, позволяет управлять формой представления окна запускаемого приложения. Эта функция может работать в любых версиях Windows и выполнять любые файлы: приложения Windows, MS-DOS, файлы PIF и т.п. Функция **WinExec** определяется следующим образом:

```
int WinExec(const char *CmdLine, unsigned int CmdShow);
```

Параметр **CmdLine** является указателем на строку с нулевым символом в конце, содержащую имя выполняемого файла и, если необходимо, параметры команд-

ной строки. Если имя файла указано без пути, то Windows будет искать выполняемый файл в каталогах в следующей последовательности:

1. Каталог, из которого загружено приложение.
2. Текущий каталог.
3. Системный каталог Windows, возвращаемый функцией **GetSystemDirectory**.
4. Каталог Windows, возвращаемый функцией **GetWindowsDirectory**.
5. Список каталогов из переменной окружения **PATH**.

Параметр **CmdShow** определяет форму представления окна запускаемого приложения Windows. Возможные значения этого параметра см. в главе 15 в разделе 15.6.3.2. Чаще всего используется значение **SW_RESTORE**, при котором окно запускаемого приложения активизируется и отображается на экране. Если это окно в данный момент свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции. Для приложений не Windows, для файлов PIF и т.д. состояние окна определяет само приложение.

При успешном выполнении запуска приложения функция **WinExec** возвращает значение, большее 31. При неудаче могут возвращаться следующие значения:

Значение	Номер	Описание
0	0	Не хватает памяти или ресурсов системы
ERROR_BAD_FORMAT	11	Ошибочный файл .exe (например, не для Win32 .EXE)
ERROR_FILE_NOT_FOUND	2	Указанный файл не найден
ERROR_PATH_NOT_FOUND	3	Указанный каталог не найден

Достоинством функции **WinExec** является ее совместимость с ранними версиями Windows. Собственно для этого она и сохраняется в Win32, хотя для Win32 рекомендуется пользоваться функцией **CreateProcess** (об этой функции см. встроенную в C++Builder справку).

Приведем примеры применения **WinExec**.

Оператор

```
WinExec("file.exe", SW_RESTORE);
```

запускает программу **file.exe**. Оператор

```
WinExec("nc", SW_RESTORE);
```

запускает Norton Commander. Оператор

```
WinExec("COMMAND.COM", SW_RESTORE);
```

приводит к запуску MS-DOS.

Операторы

```
int i = WinExec(Edit1->Text.c_str(), SW_RESTORE);
if (i < 32)
    ShowMessage("Код ошибки " + IntToStr(i));
```

обеспечивают выполнение любой программы, имя которой пользователь набрал в окне редактирования **Edit1**. Поскольку первый параметр функции должен иметь тип (**char ***), а текст окна имеет тип **AnsiString**, то для приведения типов придется использовать метод **c_str()**.

В заключение приведем пример процедуры, обеспечивающей выполнение любой выбранной пользователем программы. Откройте новый проект и разместите на форме компонент **OpenDialog**, задав в нем фильтр

программы	*.exe;*.com;*.pif;*.dat
все файлы	*.*

Разместите на форме кнопку **Button** (назовите ее **BExec**), при щелчке на которой пользователь может выбрать в окне Открыть файл программу и выполнить ее. Обработчик события **OnClick** этой кнопки может иметь вид:

```
if (OpenDialog1->Execute())
{
    int i = WinExec(OpenDialog1->FileName.c_str(), SW_RESTORE);
    switch (i)
    {
        case 0: ShowMessage("Не хватает памяти или ресурсов");
                break;
        case ERROR_BAD_FORMAT:
                ShowMessage("Ошибочный файл " + OpenDialog1->FileName);
                break;
        case ERROR_PATH_NOT_FOUND:
                ShowMessage("Не найден каталог " +
                    ExtractFilePath(OpenDialog1->FileName));
                break;
        case ERROR_FILE_NOT_FOUND:
                ShowMessage("Не найден файл " + OpenDialog1->FileName);
    }
}
```

Запустите ваше приложение на выполнение и попробуйте вызывать из него различные программы Windows и MS-DOS.

6.1.4 Запуск внешней программы и открытие документа функцией ShellExecute

Более широкие возможности по сравнению с функцией **WinExec** предоставляет функция **ShellExecute**. Она может не только выполнять заданное приложение, но и открывать документ и печатать его. Под термином «открыть файл документа» понимается выполнение связанного с ним приложения и загрузка в него этого документа. Например, обычно с документами, имеющими расширение **.doc**, связан Word. В этом случае открыть файл, например, с именем **file.doc** означает запустить Word и передать ему в качестве параметра имя файла **file.doc**. Кроме описанных возможностей функция **ShellExecute** позволяет открыть указанную папку. Это означает, что будет запущена программа «Проводник» с открытой указанной папкой.

Для использования функции **ShellExecute** в модуль надо добавить директиву препроцессора

```
#include "ShellAPI.h"
```

подключающую модуль **ShellAPI**, в котором описана функция. Автоматически C++Builder эту директиву не добавляет.

Функция **ShellExecute** инкапсулирует одноименную функцию API Windows и определена следующим образом:

```
void ShellExecute(HWND Wnd, const char * Operation,
                  const char * FileName, const char * Parameters,
                  const char * Directory, unsigned int ShowCmd);
```

Параметр **Wnd** является дескриптором родительского окна, в котором отображаются сообщения запускаемого приложения. Обычно в качестве него можно просто указать **Handle**.

Параметр **Operation** указывает на строку с нулевым символом в конце, которая определяет выполняемую операцию. Эта строка может содержать текст «**open**» (открыть) или «**print**» (напечатать). Для 32-разрядных Windows определено еще одно значение: «**explore**» (исследовать) — открыть папку программой Windows «Проводник». Если параметр **Operation** равен **NULL**, то по умолчанию выполняется операция «**open**».

Параметр **FileName** указывает на строку с нулевым символом в конце, которая определяет имя открываемого файла или имя открываемой папки.

Параметр **Parameters** указывает на строку с нулевым символом в конце, которая определяет передаваемые в приложение параметры, если **FileName** определяет выполняемый файл. Если **FileName** указывает на строку, определяющую открываемый документ или папку, то этот параметр задается равным **NULL**.

Параметр **Directory** указывает на строку с нулевым символом в конце, которая определяет каталог по умолчанию.

Параметр **ShowCmd** определяет режим открытия указанного файла. Этот параметр может иметь множество различных значений, которые указаны в главе 15 в разделе 15.6.3.2. Обычно, как и для функции **WinExec**, используется значение **SW_RESTORE**, при котором окно запускаемого приложения активизируется и отображается на экране. Если это окно в данный момент свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции. Для приложений не Windows, для файлов PIF и т.д. состояние окна определяет само приложение.

Приведем примеры использования функции **ShellExecute**. Пусть вы хотите открыть файл документа с именем **file.doc**, т.е. запустить Word (обычно именно он связан с файлами **.doc**), загрузив в него указанный файл. Тогда вы можете написать:

```
ShellExecute(Handle, NULL, "file.doc", NULL, NULL, SW_RESTORE);
```

Если вы хотите не открыть, а напечатать документ, записываются аналогичные операторы, но изменяется значение параметра **Operation**:

```
ShellExecute(Handle, "print", "file.doc", NULL, NULL, SW_RESTORE);
```

Выполнение этого оператора будет протекать следующим образом. Запустится Word, связанный с файлами **.doc**, в него загрузится файл **file.doc**, затем из Word запустится печать с атрибутами по умолчанию, после чего файл **file.doc** выгрузится из Word.

Приведенный ниже оператор открывает приложение Windows «Калькулятор»:

```
ShellExecute(Handle, "open", "Calc", NULL, NULL, SW_RESTORE);
```

Следующий пример открывает папку **c:\Program Files\Borland**:

```
ShellExecute(Handle, "open", "c:\\Program Files\\Borland",  
NULL, NULL, SW_RESTORE);
```

А оператор

```
ShellExecute(Handle, "explore", "c:\\Program Files\\Borland",  
NULL, NULL, SW_RESTORE);
```

открывает программу «Проводник» с открытой папкой **c:\Program Files\Borland**.

Функция **ShellExecute** не возвращает никакого значения. Если при вызове функции произошла ошибка, то ее можно узнать, вызвав функцию **GetLastError**:

```
DWORD GetLastError(VOID)
```

Эта функция возвращает нуль, если **ShellExecute** выполнена нормально. В противном случае возвращаются коды ошибок.

Если вы сами делаете приложение, которое будет запускаться из другого вашего приложения, то в запуске приложения вы можете установить код ошибки функцией **SetLastError**:

```
VOID SetLastError(DWORD dwErrCode)
```

Заданный этой функцией код **dwErrCode** может затем прочесть ваша вызывающая функция. Это один из способов обмена информацией между вызывающим и вызываемым приложениями.

Функция **ShellExecute** автоматически отыскивает приложение, связанное с типом открываемого документа, и запускает его. Но иногда вам может захотеться самому вызвать явным образом приложение, связанное с каким-то документом, например, чтобы передать ему какие-то дополнительные параметры. Помочь в этом может функция **FindExecutable**, которая возвращает имя и путь приложения, связанного с указанным файлом. Использование этой функции, как и предыдущих, требует включения в модуль ссылки на ShellAPI.

Функция **FindExecutable** определена следующим образом:

```
void FindExecutable(const char *FileName,  
                   const char *Directory, char *Buffer);
```

Она позволяет получить имя выполняемого файла **.exe**, связанного с файлом, указанным параметром **FileName**. Параметр **Directory** определяет каталог по умолчанию. Оба параметра являются указателями на строки с нулевым символом в конце. Параметр **Buffer** является указателем на буфер в виде строки с нулевым символом в конце, в который функция заносит имя и путь приложения, связанного с файлом **FileName**.

При успешном завершении функция **FindExecutable** возвращает значение, большее 32. Если возвращено меньшее значение, это свидетельствует об ошибке. Возможные значения ошибок те же, что и для приведенных ранее функций.

Приведем пример применения функции **FindExecutable**. Операторы

```
char APchar[254];  
FindExecutable("Doc.doc", NULL, APchar);
```

приведут к тому, что в массив **APchar** будет занесено имя приложения, связанного с файлом типа **Doc.doc**, например:

```
C:\PROGRAM FILES\MICROSOFT OFFICE\OFFICE\WINWORD.EXE
```

Правда, при этом файл **Doc.doc** должен существовать в доступном каталоге.

Успешность завершения функции **FindExecutable** можно проверить с помощью описанной ранее функции **GetLastError**. Если она возвращает значение не большее 32, то значит произошла ошибка. Эту проверку могут, например, осуществить следующие операторы:

```
int i = GetLastError();  
if (i <= 32)  
    ShowMessage("Программа не найдена. Код ошибки "+IntToStr(i));
```

Выше были рассмотрены наиболее простые функции, обеспечивающие выполнение каких-то программ из вашего приложения. В C++Builder и API Windows (все приведенные выше функции относились к API Windows, хотя для единообразия изложения они были приведены в нотации, подобной C++Builder) имеется также ряд других функций, обеспечивающих более тонкое взаимодействие с запускаемыми приложениями. В частности, имеются возможности запуска нескольких параллельно выполняемых процессов.

6.2 Управление внешними приложениями

6.2.1 Определение дескриптора окна приложения

Мы рассмотрели запуск внешних приложений. Теперь рассмотрим коротко вопросы управления ими. API Windows предоставляет немало функций, позволяющих управлять окнами приложений. Все они требуют указания дескриптора соответствующего окна. Поэтому прежде всего рассмотрим способы получения этих дескрипторов.

Получить дескриптор окна можно функцией **FindWindow**, которая имеет вид:

```
HWND FindWindow(const char *lpClassName, const char *lpWindowName);
```

Параметр **lpClassName** указывает на строку с нулевым конечным символом, содержащую имя класса. Параметр **lpWindowName** указывает на строку с нулевым конечным символом, содержащую имя окна (это свойство **Caption** формы, отображаемое в полосе заголовка окна). Если этот параметр равен **NULL**, то считается, что под критерий поиска подходит любое окно указанного класса.

Если поиск прошел успешно, то функция возвращает дескриптор окна, имеющего указанное имя класса и имя окна. В противном случае возвращается **NULL**.

Эту функцию легко использовать, если вы знаете имя класса искомого окна. Например, если ваше приложение вызвало другое приложение, созданное вами самими, то вы знаете имя класса формы этого другого приложения. Тогда вы можете, например, с помощью кода

```
HWND H = FindWindow("TForm1", "Приложение 2");
```

определить дескриптор окна приложения, класс формы которого **TForm1**, а значение свойства **Caption** формы — «Приложение 2».

Если же приложение, которым вы хотите управлять, создано не вами, то текст полосы заголовка вы легко можете увидеть, выполнив его, а вот имя класса вам неизвестно. Пусть, например, вы запустили из своего приложения программу Windows «Калькулятор», чтобы пользователь смог что-то с его помощью посчитать. Как управлять в дальнейшем этим калькулятором, если требуется, например, его свернуть, закрыть и т.д.?

Одна из возможностей узнать имя класса какого-то приложения — воспользоваться поставляемой вместе с C++Builder программой WinSight 32 (файл ...\\Program Files\\Borland\\CBuilder5\\Bin\\ws32.exe). Запустите интересующее вас приложение, затем запустите WinSight 32, выполните команду Spy | Find Window и вы увидите список всех окон, зарегистрированных в данный момент в Windows. Лучше, чтобы в этот момент у вас было бы открыто не очень много окон, чтобы проще было найти среди них нужное.

В списке, который вы увидите, для каждого окна будут указаны среди прочей информации имя класса в фигурных скобках «{ }» и заголовок окна — последний элемент данных в строке каждого окна. Например, запустив «Калькулятор», вы можете с помощью WinSight 32 найти, что имя класса окна этого приложения — «SciCalc». Следовательно, определить в своем приложении дескриптор открытого приложения «Калькулятор» вы можете оператором:

```
HWND H = FindWindow("SciCalc", "Калькулятор");
```

Другой способ найти дескриптор окна — воспользоваться функцией **GetNextWindow** API Windows. Определение этой функции:

```
HWND GetNextWindow(HWND hWnd, unsigned int wCmd);
```

Она определяет дескриптор следующего или предыдущего окна в Z-последовательности. Параметр **hWnd** — дескриптор окна, от которого начинается отсчет. Параметр **wCmd** определяет направление поиска. Если **wCmd = GW_HWND-**

NEXT, то ищется следующее окно, находящееся ниже. Если **wCmd = GW_HWNDPREV**, то ищется предыдущее окно, находящееся выше.

Следующее окно в Z-последовательности — это то, которое вызывалось из указанного или к которому пользователь обращался после создания указанного окна. Если указано дочернее окно, то поиск ведется среди дочерних окон.

Если искомое окно найдено, то возвращается его дескриптор. Если следующего или предыдущего окна нет (в зависимости от значения **wCmd**), то возвращается 0.

Сейчас мы попробуем использовать функцию **GetNextWindow** для поиска окна с известным текстом. Но при этом нам потребуется еще одна функция API Windows — **GetWindowText**. Эта функция копирует текст, связанный с указанным окном, в указанный буфер. Ее объявление имеет вид:

```
int GetWindowText(HWND hWnd, char *lpString, int nMaxCount);
```

Параметр **hWnd** — дескриптор окна. Параметр **lpString** указывает на буфер, в который копируется текст. Параметр **nMaxCount** определяет максимальное число копируемых символов. Если число символов в тексте превышает эту величину, текст усекается.

Если функция выполнялась успешно, она возвращает число скопированных символов, исключая завершающий нулевой символ. Если окно не имеет полосы заголовка или текст заголовка отсутствует, или при неверном дескрипторе возвращается нуль.

Теперь мы можем решить задачу, поставленную нами в качестве примера: определить дескриптор окна выполняемого приложения «Калькулятор». Это можно сделать с помощью следующего кода:

```
HWND H = Handle;
char Pch[128];
do
{
    H = GetNextWindow(H, GW_HWNDNEXT);
    GetWindowText(H, Pch, 128);
    if (CompareText(Pch, "калькулятор") == 0)
        break;
} while (H != NULL);
if (H != NULL)
    ...
```

Первый выполняемый оператор присваивает переменной **H** значение свойства **Handle** формы вашего приложения. Далее в цикле просматриваются окна, лежащие ниже в Z-последовательности, и среди них ищется окно с текстом «Калькулятор». Для этого используется функция **CompareText**, сравнивающая без учета регистра строку, на которую указывает **Pch**, со строкой «Калькулятор». Если строки совпадают, функция **CompareText** возвращает нуль. Пользуясь тем, что C++ позволяет оперировать с целыми значениями как с булевыми, строку оператора **if** можно было бы записать и в таком виде:

```
if( ! CompareText(Pch, "Калькулятор"))
```

Окно калькулятора будет найдено, если оно получало фокус после запуска вашего приложения. Таким образом, если пользователь запустил «Калькулятор» из вашего приложения или даже если «Калькулятор» был запущен ранее или независимо от вашего приложения, дескриптор его окна будет найден.

Если цикл завершается со значением **H = NULL**, значит приложение «Калькулятор» в данный момент не открыто. Если его все-таки надо открыть, то можно для этого воспользоваться функциями, описанными в предыдущих разделах.

6.2.2 Некоторые функции API Windows для управления окнами

Теперь рассмотрим некоторые функции API Windows, позволяющие управлять приложениями.

Функция **CloseWindow** сворачивает указанное в ней окно до пиктограммы и перемещает ее в область пиктограмм экрана. Описание этой функции:

```
bool CloseWindow(HWND hWnd)
```

Параметр **hWnd** является дескриптором сворачиваемого окна. При успешном выполнении функции возвращается ненулевое значение. При аварийном завершении возвращается нуль. Например, для сворачивания окна приложения «Калькулятор», дескриптор которого мы определяли в предыдущем разделе, можно выполнить оператор

```
CloseWindow(FindWindow("SciCalc", "Калькулятор"));
```

использующий рассмотренную ранее функцию **FindWindow**.

Функция **CloseWindow** не уничтожает окно, а только сворачивает его. Для уничтожения окна можно использовать функцию **DestroyWindow**, но она не разрешает уничтожать окно из другого потока.

Функция **EnableWindow** позволяет управлять доступом к окну. Ее описание имеет вид:

```
bool EnableWindow(HWND hWnd, bool bEnable);
```

Параметр **hWnd** определяет дескриптор того окна, которое должно быть сделано доступным или недоступным. Параметр **bEnable** задает устанавливаемое значение доступности: **true** — доступно, **false** — недоступно. Функция возвращает нуль, если окно ранее было доступным, и возвращается ненулевое значение, если оно не было доступным.

Функция **EnableWindow** делает указанное в ней окно доступным или недоступным для любых действий пользователя с помощью мыши или клавиатуры. Если окно недоступно, то предусмотренные и начатые в нем процессы продолжают, но пользователь не может воздействовать на это окно. Такое окно можно использовать для демонстрации каких-то процессов пользователю, но пользователь не может его ни свернуть, ни уничтожить.

Имеется еще множество функций API Windows для управления окнами. Они позволяют переместить окно, передать на него фокус, изменить его текст и т.д. Вы можете посмотреть информацию об этих функциях в файлах справки ...\\program files\\Common Files\\Borland Shared\\MShelp\\95guide.hlp и ...\\program files\\Common Files\\Borland Shared\\MShelp\\mapi.hlp.

6.3 Сообщения Windows и их обработка

6.3.1 Обработка сообщений в приложениях C++Builder

Выше мы рассмотрели некоторые функции API Windows, работающие с окнами. Все они неявно использовали различные сообщения Windows. Но взаимодействие с другими приложениями, выполняющимися одновременно с вашим, может быть и явным образом организовано с помощью сообщений Windows.

Приложения Windows состоят из множества объектов, которые взаимодействуют друг с другом, обмениваясь сообщениями (messages). Источниками этих сообщений могут быть: пользователь, оперирующий с клавиатурой и мышью, среда Windows, посылающая сообщения приложениям, другие приложения, обмени-

вающиеся информацией с вашим приложением и, наконец, ваше приложение, посылающее сообщения компонентам.

Большинство сообщений, которые вам могут потребоваться, C++Builder обрабатывает сам, так что вам достаточно использовать обработчики стандартных событий компонентов. Но иногда вам может потребоваться самому обрабатывать сообщения Windows. Такая необходимость возникает, если нужное вам сообщение пока еще компонентами C++Builder не обрабатывается, или если вы определили свое собственное сообщение.

Сообщение Windows представляет собой структуру, содержащую поля. Наиболее важное из них содержит целое значение, идентифицирующее данное сообщение. В C++Builder содержится объявление множества идентификаторов, позволяющие оперировать с мнемоническими именами сообщений, а не с какими-то целыми значениями. Важная информация о сообщении содержится также в двух полях параметров и в поле результата. Раньше ссылки на параметры осуществлялись как на **wParam** (word parameter — параметр типа **word**) и **lParam** (long parameter — параметр типа **long**). В ряде случаев каждый параметр содержал несколько данных, на которые надо было ссылаться, например, как на **lParamHi** — старший разряд параметра **lParam**.

Сейчас Microsoft ввел для параметров имена, что существенно упрощает работу с ними. В C++Builder также введены мнемонические имена параметров. Так что теперь, например, при обработке сообщения от мыши можно ссылаться на понятные параметры **XPos** и **YPos**, а не на стандартные и ни о чем не говорящие имена **lParamLo** и **lParamHi**.

Последовательность обработки сообщений Windows следующая. При создании объекта оконного компонента — потомка **TWinControl**, он регистрируется в Windows и получает уникальный идентификатор — дескриптор (**handle**). Вы можете получить к нему доступ через свойство **Handle** (только для чтения). При регистрации окно передает Windows указатель на процедуру, которая будет вызываться при получении им сообщений от Windows. Это метод **MainWndProc**. Его назначение — вызов метода обработки сообщений через свойство **WindowProc** и обработка исключений, которые могут возникнуть при вызове **WindowProc**. По умолчанию в свойстве **WindowProc** хранится указатель на функцию **WndProc**. Стандартная функция **WndProc** представляет собой оператор **switch**, анализирующий идентификатор сообщения и производящий соответствующие действия. Разработчик приложения может во время выполнения заменить эту стандартную функцию на свою собственную. В итоге все сообщения передаются методу **Dispatch**, который просматривает таблицу методов класса объекта и извлекает из нее тот, который имеет требуемый идентификатор сообщения. Если требуемый метод не найден ни в данном классе, ни в классах-предках, то вызывается обработчик сообщения по умолчанию **DefaultHandler**.

Таким образом, обработка сообщения Windows происходит по цепочке:

событие \Rightarrow **MainWndProc** \Rightarrow **WndProc** \Rightarrow **Dispatch** \Rightarrow обработчик события

Для сообщений, обрабатываемых компонентами C++Builder, вам достаточно написать свой обработчик события. Для сообщений, не предусмотренных в C++Builder, вам надо вмешаться в эту цепочку раньше, перегрузив метод **WndProc**.

Если объект не имеет дескриптора, то цепочка остается той же, только сообщение получает оконный компонент, вмещающий данный объект (например, форма).

В Windows предусмотрено множество сообщений. Описание некоторых из них приводится в главе 15 в разделе 15.8.2. Для дальнейших экспериментов нам потребуется только два сообщения. Первое из них — **WM_CLOSE**, сигнализирующее, что окно или приложение закрывается. Это сообщение не имеет параметров. По

умолчанию оно уничтожает окно, которому послано. Если приложение обрабатывает это сообщение, то оно должно возвращать нуль.

При обработке данного сообщения приложение может запросить пользователя о необходимости закрывать окно и вызвать функцию закрытия окна только при положительном ответе.

Второе сообщение, которое мы будем использовать — **WM_ACTIVATE**. Оно посылается, когда окно переводится в активное или неактивное состояние. Сначала сообщение посылается окну, переходящему в неактивное состояние, а потом — активизируемому.

Это сообщение определено следующим образом:

```
WM_ACTIVATE
fActive = LOWORD(wParam);
fMinimized = (BOOL) HIWORD(wParam);
hwndPrevious = (HWND) lParam;
```

Параметры этого сообщения означают следующее.

Параметр **fActive** показывает, как активируется или деактивируется окно. Возможные значения параметра:

WA_ACTIVE	окно активируется не щелчком мыши (например, функцией SetActiveWindow или клавиатурой)
WA_CLICKACTIVE	окно активируется щелчком мыши
WA_INACTIVE	окно деактивируется

Параметр **fMinimized** показывает, свернуто окно, или нет. Ненулевое значение соответствует свернутому окну.

Параметр **hwndPrevious** — это дескриптор, который указывает на окно, из которого фокус переключился на данное окно, если оно активируется, или на окно, в которое передается управление, если данное окно деактивируется.

По умолчанию, если активируемое окно не свернуто, то оно получает фокус. Если приложение обрабатывает это сообщение, то оно должно возвращать нуль.

6.3.2 Посылка сообщений

В API Windows определен ряд функций, позволяющих послать сообщение.

6.3.2.1 Функции **SendMessage**, **PostMessage** и **Perform**

Функция **SendMessage** посылает указанное в ней сообщение окну или множеству окон и не возвращается, пока это сообщение обрабатывается. Этим она отличается от функции **PostMessage**, которая возвращается сразу после передачи сообщения.

Объявление функции **SendMessage**:

```
Int SendMessage(HWND hWnd, unsigned int Msg,
                 WPARAM wParam, LPARAM lParam);
```

Параметр **hWnd** — дескриптор окна, которому передается сообщение. Если этот параметр равен **HWND_BROADCAST**, то сообщение передается всем окнам верхнего уровня в системе, включая недоступные и невидимые, кроме дочерних.

Параметр **Msg** определяет передаваемое сообщение. Параметры **wParam** и **lParam** могут содержать дополнительную информацию. Значение, возвращаемое функцией, зависит от вида сообщения.

Функции **PostMessage** объявлена следующим образом:

```
bool PostMessage(HWND hWnd, unsigned int Msg,  
                WPARAM wParam, LPARAM lParam);
```

Эта функция похожа на **SendMessage**, но в отличие от нее она помещает сообщение в очередь и сразу возвращается. Таким образом, **PostMessage** не годится для передачи срочных сообщений, но зато она не блокирует вызвавшее приложение на время обработки сообщения приемником.

Параметры **hWnd** и **Msg** аналогичны рассмотренным для функции **SendMessage**. Если **hWnd = NULL**, то сообщение ставится в очередь сообщений (если она есть) текущего процесса.

Функция **PostMessage** возвращает ненулевое значение при успешном завершении и нуль в случае аварийного завершения. В этом случае причину ошибки можно установить вызовом функции **GetLastError**.

Имеется еще один метод, который может посылать сообщение непосредственно оконному компоненту. Это метод **Perform**, объявление которого имеет вид:

```
Perform(unsigned int Msg, WPARAM wParam, LPARAM lParam);
```

Есть еще ряд функций, позволяющих передавать сообщения, но мы на них не будем останавливаться, так как они реже используются в приложениях C++Builder.

6.3.2.3 Пример отправки сообщений

Давайте построим простую программу, демонстрирующую отсылку сообщений. Начните новый проект, создайте в нем две формы **Form1** и **Form2**, сохраните проект, назвав модули форм **U1Mess1** и **U2Mess1** соответственно, а файл проекта — **PMess1**. Форма **Form1** будет у нас главной и она будет управлять видимостью формы **Form2**. Поэтому в ее модуль введите директиву препроцессора

```
#include "U2Mess1.h"
```

а свойство **Visible** формы **Form2** должно быть равно **false**.

Перенесите на форму **Form1** две кнопки, дав им надписи «Show Form2» и «Close Form2». В обработчике щелчка первой кнопки напишите оператор

```
Form2->Show();
```

а в обработчике щелчка второй кнопки — оператор

```
SendMessage(Form2->Handle, WM_CLOSE, 0, 0);
```

Этот оператор посылает сообщение **WM_CLOSE** (второй параметр функции **SendMessage**) форме **Form2**. Первый параметр функции **SendMessage** содержит дескриптор окна этой формы, полученный с помощью ее свойства **Handle**. Сообщение **WM_CLOSE** не имеет параметров; поэтому параметры **wParam** и **lParam** заданы равными нулю.

Приведенный выше оператор можно заменить на следующий:

```
Form2->Perform(WM_CLOSE, 0, 0);
```

Результат будет тем же самым.

Сохраните и запустите приложение. Вы увидите, что нажатие пользователем кнопки **Show Form2** приводит к появлению на экране формы **Form2**, а нажатие кнопки **Close Form2** — к ее закрытию. Так что посылаемое формой **Form1** сообщение достигает своего адресата.

Конечно, этот пример не очень интересный, поскольку сделать невидимой форму **Form2** мы могли бы и не посылая никаких сообщений Windows, а просто используя метод **Hide**. Поэтому пойдем дальше. Откройте новый проект, задайте заголовок **Caption** ее формы равным «Приложение **PMess2**» (этот текст мы будем далее использовать для идентификации окна этого приложения) и сохраните проект под именем **PMess2**, а модуль формы — под именем **U1Mess2**. Кстати, поскольку у нас будет два проекта, можете включить их в общую группу, чтобы опробовать

вать этот инструмент (см. раздел 2.4.3). Откомпилируйте новый проект и сохраните. В дальнейшем мы еще к нему вернемся.

Теперь давайте попробуем управлять этим проектом из формы **Form1** проекта **PMess1**. Откройте опять проект **PMess1** и добавьте на форму **Form1** две кнопки, сделав на них надписи «Exec Pmess2» и «Close Pmess2». В обработчике щелчка первой кнопки напишите оператор

```
WinExec("Pmess2.exe", SW_RESTORE);
```

который, как вы уже знаете, запускает приложение **PMess2** на выполнение. А теперь давайте попробуем его закрыть. Для этого в обработчике щелчка кнопки **Close Pmess2** напишите оператор

```
SendMessage(FindWindow("TForm1", "Приложение Pmess2"), WM_CLOSE, 0, 0);
```

Этот оператор использует функцию **FindWindow** для получения дескриптора окна приложения, которому надо послать сообщение, а затем функцией **SendMessage** посылает сообщение **WM_CLOSE**.

Сохраните и запустите приложение. Теперь, нажимая кнопку **Exec Pmess2**, вы можете выполнять приложение **PMess2**, причем можете создать несколько экземпляров этого приложения. А кнопкой **Close Pmess2** можете закрывать приложение **PMess2**.

Вы получили возможность управлять из своего приложения другими. Но в данном случае вы знали класс окна (**TForm1**) внешнего приложения, поскольку вы сами его создавали. Поэтому вы смогли применить функцию **FindWindow**, передав в нее и имя класса, и заголовок окна. А как быть, если вы хотите закрыть какое-то чужое приложение? Например, вы из своего приложения выполнили стандартное приложение Windows «Калькулятор», пользователь посчитал, что ему было надо, а теперь вы хотите закрыть «Калькулятор», послав ему сообщение **WM_CLOSE** (например, пользователь уже не работает с ним, а закрыть забыл).

При посылке сообщения другому приложению возникает задача определить дескриптор нужного окна. Как это можно сделать обсуждалось ранее в разделе 6.2.1. Если вы определили имя класса необходимого вам приложения (например, **SciCalc** для приложения «Калькулятор») с помощью **WinSight 32** и хотите послать из своего приложения сообщение о закрытии калькулятора, вы можете выполнить оператор:

```
SendMessage(FindWindow("SciCalc", "калькулятор"), WM_CLOSE, 0, 0);
```

Если вы определили дескриптор окна способами, изложенными в 6.2.1, то вы просто посылаете сообщение окну с данным дескриптором.

6.3.3 Обработка сообщений

Во всех оконных компонентах предусмотрены обработчики сообщений Windows по умолчанию. До сих пор вы пользовались именно стандартными обработчиками сообщений. Однако, вы можете определить и свои собственные обработчики, заменив ими обработчики по умолчанию, или дополнив их.

Для введения собственного обработчика сообщения Windows надо сделать следующее:

1. Создать в объявлении класса карту (map) сообщений и ввести в нее те сообщения, которые вы хотите обрабатывать сами.
2. Добавить в объявление класса объявления вводимых вами обработчиков.
3. Описать эти обработчики в вашем модуле.

Первый шаг — объявление карты сообщений, легко осуществляется с помощью следующих макросов:

```
BEGIN_MESSAGE_MAP
```

```
MESSAGE_HANDLER(parameter1, parameter2, parameter3)
...
END_MESSAGE_MAP
```

Макрос **BEGIN_MESSAGE_MAP** открывает объявление карты сообщений, макрос **END_MESSAGE_MAP** завершает это объявление, а один или несколько макросов **MESSAGE_HANDLER** вводят в карту соответствующие сообщения. В макросе **MESSAGE_HANDLER** первый параметр указывает имя сообщения, второй — тип структуры сообщения, а третий — имя функции-обработчика.

Имя сообщения пишется заглавными буквами и должно соответствовать предопределенному в Windows сообщению. Например, **WM_CLOSE**. Имя типа структуры сообщения может быть любым, но обычно принято делать его тождественным имени обрабатываемого сообщения с исключенным из него символом подчеркивания и добавленным префиксом **T**. Например, **TWMClose**. Передаваемый в обработчик параметр этого типа представляет собой структуру, через которую в обработчик передаются параметры сообщения, а из обработчика возвращается значение поля **Result**, фиксирующее результат обработки. Имя функции-обработчика сообщения также может быть любым, но обычно оно образуется из имени сообщения исключением первых символов **WM_** и добавлением префикса **On**. Например, **OnClose**.

Давайте введем в рассмотренное ранее наше приложение **PMess2** и в форму **Form2** приложения **PMess1** обработку тех сообщений **WM_CLOSE**, которые мы посылаем им из формы **Form1**. Для этого поместим на форму **Form1** приложения **PMess2** и на форму **Form2** приложения **PMess1** метки **Label1**, чтобы отображать в них текст, свидетельствующий о том, что обработка сообщения действительно происходит. Введем в модулях этих форм следующие обработчики (текст приведен для **Form2**; для **Form1** все то же самое с заменой идентификатора **Form2** на **Form1**):

```
// модуль U2Mess1.h
class TForm2 : public TForm
{
__published:      // IDE-managed Components
    TLabel *Label1;
private:          // User declarations
    void __fastcall OnClose(TWMClose& Message);
public:           // User declarations
    __fastcall TForm2(TComponent* Owner);
    BEGIN_MESSAGE_MAP
        MESSAGE_HANDLER(WM_CLOSE, TWMClose, OnClose)
    END_MESSAGE_MAP(TComponent)
};
//-----
// модуль U2Mess1.cpp
...
void __fastcall TForm2::OnClose(TWMClose& a)
{
    Label2->Caption = "Караул! Закрывают!";
    if (MessageDlgPos("Меня хотят закрыть. Согласны?",
        mtConfirmation, TMsgDlgButtons() << mbYes << mbNo << mbCancel,
        0, BoundsRect.Left, BoundsRect.Bottom) == mrYes)
        Close();
    else Label2->Caption = "Не закроюсь!";
    a.Result = 0;
}
```

В этом коде в файле **U2Mess1.h** в разделе **public** объявлена карта сообщений, в которую включено сообщение **WM_CLOSE**. Для этого сообщения объявлен обработчик **OnClose** и тип передаваемого в него параметра — **TWMClose**. В разделе

private объявлена функция этого обработчика. В нее передается параметр, названный **message** — структура сообщения.

В обработчике **OnClose** сообщения **WM_CLOSE** с параметром, которому дано имя **a**, производится запрос подтверждения пользователя о закрытии окна. Для запроса использована процедура **MessageDlgPos**, позволяющая указать позицию окна запроса вблизи окна формы, к которой относится запрос. При положительном ответе пользователя окно закрывается методом **Close**. В заключение оператором **a.Result = 0** возвращается нуль, так как это действие оговорено в приведенном ранее описании сообщения **WM_CLOSE**.

Откомпилируйте оба ваших приложения и выполните приложение **PMess1**. После того, как вы сделаете кнопкой **Show Form2** видимой форму **Form2** и запустите кнопкой **Exec Pmess2** приложение **PMess2**, вы увидите, что окна формы **Form2** и **PMess2** оказывают сопротивление попыткам их зарыть, независимо от того, как это делается: кнопками **Close Form2** и **Close Pmess2**, или кнопками в полосах заголовков этих окон. В любом случае они закрываются только после подтверждения пользователя.

Кстати, вы можете наглядно посмотреть и очередь сообщений. Щелкните на кнопке **Close Pmess2** и, пока на экране отображено окно запроса, вернитесь, ничего не отвечая, в окно формы **Form1** первого приложения и щелкните еще несколько раз на кнопке **Close Pmess2**. После этого ответьте на запрос отрицательно. Вы увидите, что окно запроса без всяких дополнительных действий с вашей стороны будет отображаться еще столько раз, сколько раз вы щелкнули перед этим на **Close Pmess2**. Все сообщения, связанные с этими щелчками, запомнились в очереди и теперь очередь разгружается.

Приведенный пример обработки сообщений не очень показательный, поскольку сообщение **WM_CLOSE** не имеет параметров. Давайте усовершенствуем наши приложения так, чтобы поработать с сообщениями, имеющими параметры. Воспользуемся для этого описанным ранее (раздел 6.3.1) сообщением **WM_ACTIVATE**. Это сообщение получает любое окно при его активации или деактивации. Сообщение имеет, в частности, параметр **fActive**. Значение **WA_INACTIVE** этого параметра показывает, что окно деактивируется. Иные значения параметра показывают, что окно активируется.

Давайте введем во все наши формы (**Form1** и **Form2** приложения **PMess1**, и **Form1** приложения **PMess2**) обработчики сообщения **WM_ACTIVATE**. Чтобы видеть, что эти обработчики работают правильно, поместим на эти формы дополнительно метки **Label2**, в которых будем отображать результаты обработки. И во все три формы введем обработчики вида (приводится текст для формы **Form2**):

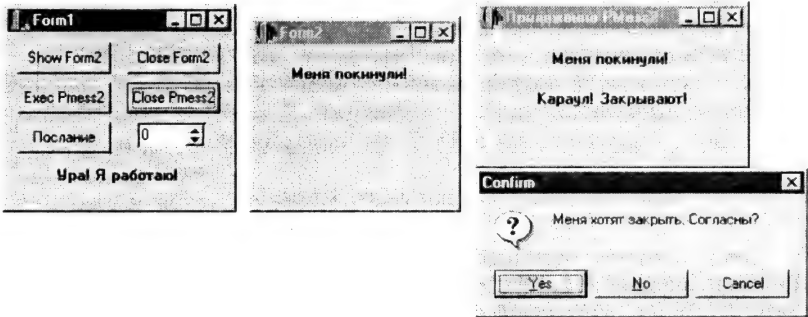
```
// модуль U2Mess1.h
class TForm2 : public TForm
{
__published:      // IDE-managed Components
    TLabel *Label1;
    TLabel *Label2;
private:           // User declarations
    void __fastcall OnClose(TWMClose& Message);
    void __fastcall OnActivate(TWMActivate& Message);
public:            // User declarations
    __fastcall TForm2(TComponent* Owner);
    BEGIN_MESSAGE_MAP
        MESSAGE_HANDLER(WM_CLOSE, TWMClose, OnClose)
        MESSAGE_HANDLER(WM_ACTIVATE, TWMActivate, OnActivate)
    END_MESSAGE_MAP(TComponent)
};
//-----
// модуль U2Mess1.cpp
...
void __fastcall TForm2::OnActivate(TWMActivate& a)
```

```
{
    if (a.Active == WA_INACTIVE)
        Label2->Caption = "Меня покинули!";
    else Label2->Caption = "Ура! Я работаю!";
    a.Result = 0;
}
```

Обработчик **OnActivate** строится по тем же принципам, что и предыдущий. В нем анализируется значение поля **Active** структуры **a**, передаваемой в процедуру в качестве параметра и содержащей параметры сообщения. В зависимости от значения этого параметра производятся те или иные действия.

Откомпилируйте оба ваших приложения и выполните приложение **PMess1**. Вы увидите (рис. 6.1), как при каждом переключении фокуса между окнами в них появляются сообщения, отражающие потерю и обретение ими активности.

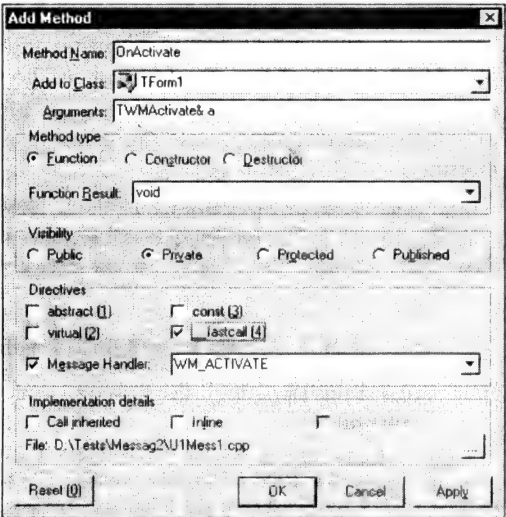
Рис. 6.1
Приложения, обменивающиеся сообщениями



Подобное приложение, дополненное еще несколькими обработчиками событий, вы можете увидеть на диске, приложенном к книге.

В C++Builder 5 создание собственных обработчиков исключений упростилось, благодаря новым возможностям Исследователя Классов ClassExplorer (раздел 2.5.3.2). Попробуйте его при создании, например, обработчика события **OnActivate**. Если окно Исследователя Классов у вас не открыто, выполните команду View | ClassExplorer. Щелкните в окне ClassExplorer правой кнопкой мыши и во всплывшем меню выберите раздел New Method — новый метод. Перед вами откроется окно, представленное на рис. 6.2.

Рис. 6.2
Диалоговое окно задания нового метода



Поскольку вы создаете обработчик сообщения, надо включить индикатор Message Handler. Затем из расположенного правее индикатора выпадающего списка вы можете выбрать имя сообщения — **WM_ACTIVATE**. В верхнем окошке Method Name вы должны написать имя создаваемого метода — **OnActivate**. В выпадающем списке Add to Class вы можете выбрать класс, объявленный в проекте, в который вводится свойство. В окошко Arguments вам надо занести список аргументов функции в том виде, в котором он должен появиться в скобках ее объявления — в нашем случае «**TWMActivate& a**». Поскольку создается функция, надо включить радиокнопку Function. При этом становится доступным выпадающий список Function Result, позволяющий задать тип возвращаемого функцией значения — для нашего примера **void**. Группа радиокнопок Visibility задает доступ к методу: **public**, **private**, **protected** или **published**. Группа индикаторов Directives позволяет задать спецификаторы объявления метода. Их смысл ясен из надписей около индикаторов.

После того, как вы установили в окне рис. 6.2 всю необходимую информацию, можете щелкнуть на ОК для выхода из окна или на Apply для внесения в код заданных установок и продолжения работы в диалоговом окне.

Вы увидите, что в заголовочном файле появятся все необходимые объявления, включая объявление карты сообщений. А в файле реализации появится код:

```
void __fastcall TForm1::OnActivate(TWMActivate& a)
{
    //TODO: Add your source code here
}
```

Это заготовка функции вашего обработчика с включенным оператором To-Do List (см. раздел 2.4.4.1). Он обеспечивает отображение соответствующей строки в окне To-Do List, которая будет напоминать вам о необходимости завершить кодирование обработчика.

Приведенный пример показывает, какую большую помощь может оказать вам ClassExplorer в C++Builder 5 при создании обработчиков сообщений.

6.3.4 Определение собственных сообщений

В предыдущих разделах посылались и обрабатывались сообщения, предопределенные API Windows. Однако, вы можете описать свои собственные сообщения и работать с ними так же, как с предопределенными.

Номера своих собственных сообщений вы должны отсчитывать от константы **WM_USER**, которая соответствует первому номеру сообщения пользователя.

Например, вы можете определить в своем приложении константы

```
#define WM_MyMess1 WM_USER
#define WM_MyMess2 WM_USER + 1
```

и затем оперировать с сообщениями **WM_MyMess1** и **WM_MyMess2** как с предопределенными в Windows. Например, можете вставить в карту сообщений объявление:

```
MESSAGE_HANDLER(WM_MyMess1, TMessage, OnMyMess1)
```

В данном объявлении в качестве типа параметра использован тип **TMessage**. Этот тип определяет следующую структуру:

```
struct TMessage
{
    unsigned int Msg;
    long WParam;
    long LParam;
    long Result;
};
```

Вы можете при посылке сообщения передавать параметрами **WParam** и **LParam** любую необходимую информацию.

В качестве примера давайте добавим в нашем тестовом приложении **PMess1** на форму **Form1** компонент **CSpinEdit** и кнопку, задав ей надпись **Послание**. В обработчик щелчка на этой кнопке мы хотим вставить посылку сообщения второму нашему приложению — **PMess2**, в котором в качестве кода послания передать число, установленное пользователем в компоненте **CSpinEdit**.

Для того, чтобы сделать это, объявите в заголовочном файле **U1Mess1.h** номер вашего сообщения с именем, например, **WM_MyPost**:

```
#define WM_MyPost WM_USER
```

а в обработчик события щелчка на кнопке **Послание** вставьте оператор

```
SendMessage(FindWindow("TForm1", "Приложение Pmess2"),
            WM_MyPost, 0, CSpinEdit1->Value);
```

Вот и все! Сообщение будет посылаться. Можно было даже сделать проще: не вводить константы, а просто в функции **SendMessage** указать вместо **WM_MyPost** номер сообщения — **WM_USER**. Но с именем сообщения код читается проще.

Теперь осталось написать обработчик этого сообщения в приложении **PMess2**. Это выглядит так же, как и для других обработчиков сообщений:

```
// модуль U1Mess2.h
...
#define WM_MyPost WM_USER
//-----
class TForm1 : public TForm
{
    ...

private: // User declarations
    void __fastcall OnMyPost(TMessage& Message);
public: // User declarations
    __fastcall TForm1(TComponent* Owner);
    BEGIN_MESSAGE_MAP
        ...
        MESSAGE_HANDLER(WM_MyPost, TMessage, OnMyPost)
    END_MESSAGE_MAP(TComponent)
};
//-----

// модуль U1Mess2.cpp
...
void __fastcall TForm1::OnMyPost(TMessage& a)
{
    Label2->Caption = "Получено письмо " + IntToStr(a.LParam);
}
```

В объявлении и реализации обработчика использован тип **TMessage** — стандартный тип параметра сообщения Windows.

Можете запускать приложения и убедиться, что послание нормально передается и воспринимается. Конечно, это просто демонстрация возможностей обмена собственными сообщениями. В реальных приложениях можно по номеру параметра оператором **switch** выбирать тот или иной вид реакции приложения на полученное сообщение.

6.4 Внедрение и связывание объектов — OLE

6.4.1 Общие сведения

Прежде, чем рассматривать реализацию в C++Builder технологии *внедрения и связывания объектов OLE* (произносится «оле» с ударением на последний слог), остановимся на некоторых базовых понятиях. Эта технология появилась как OLE

1.0 в Windows 3.1 и означала, что пользователь мог создавать сложные составные документы, в которых содержались объекты различного происхождения. *Внедренные объекты* могли редактироваться простым двойным щелчком мыши на соответствующем элементе данных. Например, можно было дважды щелкнуть на электронной таблице Excel, встроенной в документ редактора Word, и в отдельном окне запускался Excel с загруженным рабочим листом, готовым к редактированию. После завершения редактирования Excel позволял сохранить изменения во внедренном в документ Word объекте Excel.

Другой особенностью было *связывание объектов*. Это позволяло связать электронную таблицу с документом Word (по сути внутри документа Word хранился просто указатель на электронную таблицу). Если данные в оригинале электронной таблицы обновлялись, то при следующей загрузке документа Word ссылка обновляла документ и отражала в нем проведенные изменения.

Дальнейшее развитие внедрение и связывание получило в OLE 2.0. Основой этого усовершенствованного подхода явилась *компонентная модель объекта (COM)*. Это модель объекта в системном обеспечении, которая предусматривает полную совместимость во взаимодействии между компонентами, написанными разными компаниями и на разных языках. Ключом к успеху является модульность этих компонентов. Они могут покупаться, модернизироваться или заменяться поодиночке или группами, причем это никак не влияет на работу целого.

COM определяет двоичный интерфейс, полностью независимый от языка программирования, использованного при реализации компонента. Компонент, написанный в соответствии со спецификациями двоичного интерфейса COM, может вступать во взаимодействие с другим компонентом, не зная в действительности ничего о реализации последнего.

Новая особенность, появившаяся в OLE 2.0, — это *автоматизация OLE*, которая обеспечивает доступ к объектам приложения и манипуляцию с ними извне. Такие объекты, предоставленные (экспонированные) для внешнего пользования, называются *автоматными объектами OLE*. Типы объектов, которые могут быть экспонированы, так же разнообразны, как и сами приложения Windows. Текстовый процессор может экспонировать в качестве автоматного объекта документ, абзац или предложение. Электронная таблица может экспонировать таблицу, диаграмму, ячейку или группу ячеек.

Главное отличие автоматных объектов от обычных объектов OLE состоит в том, что автоматные объекты доступны только программно, они создаются и используются при помощи программного кода и, следовательно, в принципе временны. Они не могут быть внедрены или связаны. Они могут существовать только в течение времени выполнения ваших программ и не видны непосредственно конечному пользователю.

Существуют два типа автоматных серверов: внутри-процесса и вне-процесса (их еще называют локальными). C++Builder поддерживает оба типа серверов.

Сервер внутри процесса является DLL (динамически связываемой библиотекой), которая экспортирует автоматные объекты. Поскольку автоматные объекты поставляются из DLL, а не из других приложений, они являются частью приложения клиента. Это избавляет от больших накладных расходов, сопутствующих каждому вызову автоматного сервера.

Сервер вне-процесса — это автономный исполняемый файл, экспортирующий автоматные объекты. Примером этого мог бы быть Microsoft Word. Word имеет некоторое количество объектов, которые он экспонирует в качестве автоматных. Позже будет приведен пример использования Word как автоматного сервера.

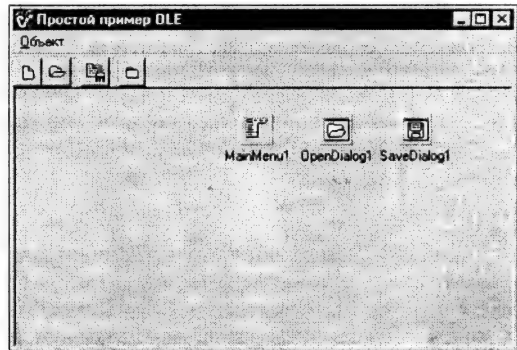
6.4.2 Внедрение и связывание

Теперь рассмотрим основные приемы использования OLE в C++Builder. На странице System библиотеки визуальных компонентов имеется контейнер OLE **OleContainer** — компонент, обеспечивающий внедрение и связывание. Давайте посмотрим его возможности сначала на очень простом примере. Разместите на форме контейнер **OleContainer**, компонент главного меню **MainMenu**, диалоги **OpenDialog** и **SaveDialog**. Можно также разместить панель и на ней быстрые кнопки, дублирующие основные команды меню. Но меню, как мы увидим ниже, должно быть обязательно.

Панель, если вы ее ввели, выравнивается по верху формы (**Align = alTop**), а контейнер должен занимать всю оставшуюся площадь формы (**Align = alClient**). Пример такого приложения приведен на рис. 6.3.

Рис. 6.3

Форма простого приложения OLE



В **MainMenu** введите меню **Объект** и в нем разделы **Новый**, **Открыть**, **Сохранить**, **Заккрыть**. Назовите объекты этих разделов **MNew**, **MOpen**, **MSave** и **MClose**.

Установите в диалогах в свойстве **Filter**: «объекты OLE | *.ole» и «все файлы | *.*». Установите также в диалогах расширение по умолчанию в свойстве **DefaultExt** равным **ole**.

Теперь перейдем собственно к программированию. Давайте введем в определении класса формы глобальную переменную **FName**, в которой будет храниться имя файла объекта OLE:

```
AnsiString FName;
```

Процедура, соответствующая разделу меню **Новый**, может иметь вид:

```
{
    if (OleContainer1->InsertObjectDialog())
        FName = "";
}
```

Вызываемый этим оператором метод **InsertObjectDialog** осуществляет обращение к стандартному окну Windows **Insert Object** (вставка объекта), в котором пользователь может указать тип вставляемого объекта, инициализирует объект OLE и загружает его в контейнер **OleContainer1**. Работу с окном **Вставка объекта** мы рассмотрим несколько позднее, а пока продолжим создание приложения.

Процедура, соответствующая разделу меню **Заккрыть**, может иметь вид:

```
void __fastcall TForm1::MCloseClick(TObject *Sender)
{
    OleContainer1->DestroyObject();
}
```

Эта процедура разрушает объект в контейнере OLE.

Процедура, соответствующая разделу меню **Сохранить**, может иметь вид:

```
void __fastcall TForm1::MSaveClick(TObject *Sender)
{
    if (FName == "")
        if (SaveDialog1->Execute())
            FName = SaveDialog1->FileName;
        else return;
    OleContainer1->SaveToFile(ChangeFileExt(FName, ".ole"));
}
```

Если имя файла не задано, то вызывается диалог Сохранить как (**SaveDialog1**), с помощью которого пользователь задает имя файла. Затем методом **SaveToFile** объект сохраняется в файле. При этом во избежание ошибок расширение файла принудительно заменяется на **.ole** с помощью функции **ChangeFileExt**, изменяющей расширение файла (см. главу 15 раздел 15.5.5).

Процедура, соответствующая разделу меню Открыть, может иметь вид:

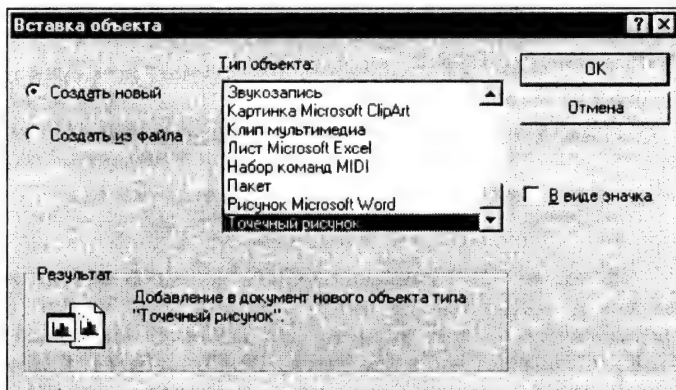
```
void __fastcall TForm1::MOpenClick(TObject *Sender)
{
    if (OpenDialog1->Execute())
    {
        OleContainer1->LoadFromFile(OpenDialog1->FileName);
        FName = OpenDialog1->FileName;
        OleContainer1->Repaint();
    }
}
```

Она обычным образом вызывает диалоговое окно, в котором пользователь выбирает открываемый файл. Затем объект, соответствующий этому файлу, загружается в контейнер методом **LoadFromFile**. Имя файла запоминается в переменной **FName**. Последний оператор методом **Repaint** перерисовывает окно контейнера, чтобы в нем немедленно отобразился открытый объект.

Теперь посмотрим, как работает наше приложение. Запустите его на выполнение и выполните команду Новый. Перед вами откроется окно Вставка объекта (Insert Object), представленное на рис. 6.4. В этом окне вам предоставляются две возможности: создание нового объекта OLE (радиокнопка Создать новый), или создать объект из имеющегося файла (радиокнопка Создать из файла).

Рис. 6.4

Окно вставки объекта OLE — вставка нового объекта

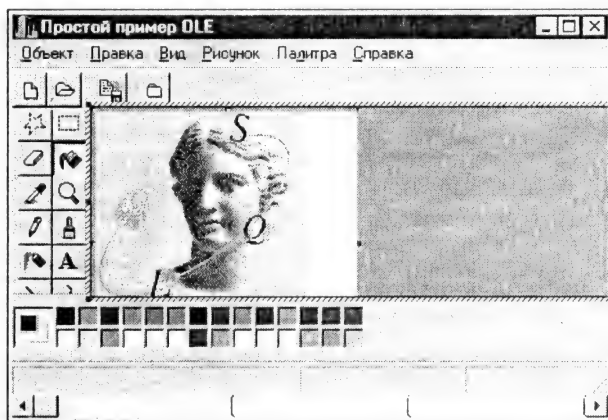


Рассмотрим сначала первую возможность. А этом случае в окне Тип объекта вы должны указать тип вставляемого в контейнер объекта. Это может быть документ Word, лист Excel, объект звукозаписи, точечный рисунок (как на рис. 6.4) и т.п. Правда, к сожалению, надо отметить, что не любой тип объекта и не в любой версии Windows может быть вставлен. Так что с переносимостью подобных приложений могут возникнуть проблемы. Если вставка не получится, вам будет выдано соответствующее сообщение.

После того, как вы выбрали тип объекта и щелкнули на ОК, в контейнере OLE вашего приложения отобразится вид объекта по умолчанию. Этот вид зависит от типа объекта. Возможно, в первый момент вы вообще ничего не увидите. Сделайте двойной щелчок на контейнере, вызывая тем самым программу, обслуживающую объект этого типа. Вы увидите, что ваше приложение чудесным образом преобразится. В него встроится соответствующая программа вместе со своими инструментальными панелями и меню. На рис. 6.5 вы видите встроенную программу Microsoft Paint, обслуживающую объекты битовых матриц — точечные рисунки. Вы можете нарисовать какое-то свое изображение или с помощью команды ее меню Правка | Вставить из файла открыть какой-то файл с битовой матрицей. На рис. 6.5, например, открыт файл, расположенный в каталоге \program files\Common Files\Borland Shared\Images\Splash\16Color\athena.bmp. Далее вы можете редактировать загруженное изображение (правда, надеюсь, у вас не поднимется рука редактировать показанное на рис. 6.5 прекрасное фирменное изображение Borland).

Рис. 6.5

Приложение с внедренным объектом OLE — редактирование объекта



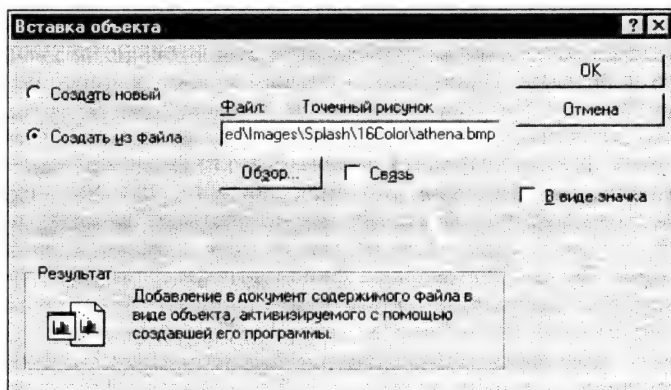
Обратите внимание на то, что ваша инструментальная панель сохранилась на экране и после загрузки объекта и начала работы с ним. И, главное, заметьте, что не все разделы меню программы Microsoft Paint встроились в ваше приложение. Нет первого меню — Файл, в котором содержатся команды открытия и сохранения файлов. Это не случайно, так как далее будет показано, что сохранение объекта OLE отличается от сохранения файла изображения. Как вы можете видеть на рис. 6.5, вместо меню Файл, имеющегося в Microsoft Paint, встроилось меню Объект вашего приложения, в котором имеются соответствующие команды открытия и закрытия файла. Именно ради этого мы и делали в своем приложении меню.

Мы рассмотрели одну из возможностей создания в вашем приложении объекта OLE. Теперь давайте вернемся к окну рис. 6.4 и рассмотрим другие возможности. Выполните опять команду Новый вашего приложения и в окне рис. 6.4 выберите радиокнопку Создать из файла (Create from File). В этом случае вы можете создать объект OLE на основе имеющегося файла. При этом диалоговое окно изменит свой вид (см. рис. 6.6). В нем вы можете с помощью кнопки Обзор выбрать какой-нибудь файл, например, тот же файл **athena.bmp**, который использован в предыдущем примере. Обратите внимание на очень важный индикатор Связь (Link). Пока не устанавливайте этот индикатор, а просто нажмите ОК.

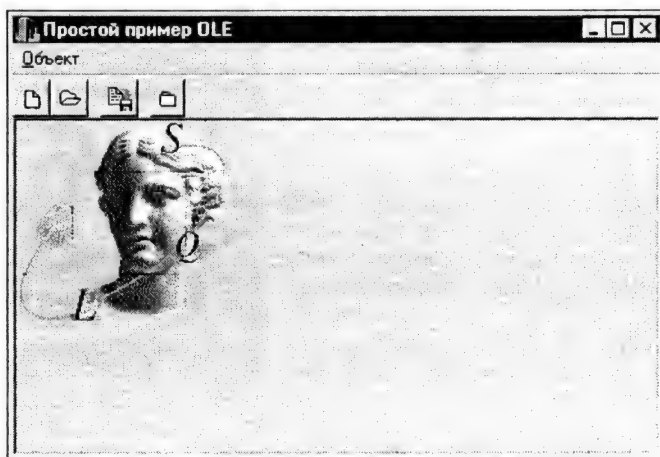
В контейнер вашего приложения загрузится содержимое файла (рис. 6.7). Вы создали в своем приложении внедренный (но не связанный) объект OLE. Если вы хотите вызвать программу, обслуживающую этот объект (в нашем примере — Microsoft Paint), сделайте на нем двойной щелчок. Вы увидите ту же картину, с которой уже знакомы по рис. 6.5 — программа Microsoft Paint встроится в ваше приложение.

Рис. 6.6

Окно вставки объекта OLE —
вставка объекта из файла

**Рис. 6.7**

Приложение с внедренным
объектом OLE — просмотр
файла



Теперь остановимся на сохранении объекта. Если вы выполните команду вашего приложения Сохранить, то с помощью обычного диалога сохранения можете записать объект в нужный вам каталог. Тем самым вы создадите файл, содержащий внедренный объект OLE. В дальнейшем вы можете открыть его командой Открыть вашего приложения. Это абсолютно автономный объект, никак не связанный с исходным файлом, из которого он был создан. Но открыть его можно только как объект OLE. Если вы попытаете открыть его, например, программой Microsoft Paint, вы потерпите неудачу, так как Microsoft Paint не поймет формата этого файла.

Теперь попробуйте создать внедренный и связанный документ. Выполните опять команду Новый вашего приложения, в диалоговом окне Вставка объекта, представленном на рис. 6.4, опять выберите радиокнопку Создать из файла (Create from File), затем в окне рис. 6.6 выберите какой-нибудь файл, но на этот раз установите индикатор Связь (Link). После щелчка на ОК в окне вашего приложения снова появится выбранный вами объект документа. Вы опять можете сохранить этот объект командой Сохранить. Но теперь документ в вашем объекте не просто внедрен, а и связан с исходным файлом. Это означает, что все изменения в исходном файле отразятся в вашем объекте и наоборот. Кроме того изменяется и взаимодействие вашего приложения с документом. Если вы сделаете двойной щелчок на объекте в вашем приложении, то откроется отдельное, полноценное окно программы, обрабатывающей ваш объект, например, Microsoft Paint, с загруженным в него доку-

ментом. Вы можете что-то изменить в нем, напечатать его, сохранить. Как только он будет сохранен, в вашем приложении отразятся введенные изменения.

Вы создали очень простое приложение, использующее OLE. Его можно усовершенствовать, добавив в меню еще несколько разделов. Один из них — Открыть файл. В отличие от рассмотренного ранее раздела, открывающего объект OLE, в данном разделе можно создавать новый объект на основе какого-то существующего файла документа. Делается это методом **CreateObjectFromFile**:

```
void CreateObjectFromFile(AnsiString FileName, bool Iconic);
```

Аргумент **FileName** определяет имя открываемого файла. Второй аргумент функции **Iconic**, значение которого обычно задается равным **false**, показывает, что объект отображается в том виде, в каком он содержится в исходном файле. Если задать этот аргумент, равным **true**, то объект будет отображаться в виде пиктограммы.

Таким образом, в вашем приложении обработчик команды Открыть файл может иметь вид:

```
void __fastcall TForm1::MOpenFClick(TObject *Sender)
{
    if (OpenDialog1->Execute())
    {
        OleContainer1->CreateObjectFromFile(OpenDialog1->FileName, false);
        FName = OpenDialog1->FileName;
        OleContainer1->Repaint();
    }
}
```

Выполнение этой команды эквивалентно описанному ранее созданию внедренного объекта OLE из файла, но исключает необходимость пользователю работать с окнами рис. 6.4 и 6.6.

Имеется также аналогичный метод **CreateLinkToFile**, позволяющий создавать внедренный и связанный объект:

```
void CreateLinkToFile(const AnsiString FileName, bool Iconic);
```

Можете воспользоваться им для включения в ваше меню и такого раздела.

Еще один раздел, который можно добавить в меню вашего приложения — Сохранить файл. В отличие от рассмотренного ранее раздела, сохраняющего в файле объект OLE, в данном разделе можно сохранять документ, содержащийся в объекте, в его натуральном виде. Это можно сделать оператором, использующим функцию **SaveAsDocument**:

```
OleContainer1->SaveAsDocument(FName);
```

Функции **CreateObjectFromFile**, **CreateLinkToFile** и **SaveAsDocument** позволяют построить фактически универсальный редактор текстовых, графических, музыкальных и других файлов.

6.4.3 Автоматизация OLE

Внедрение и связывание позволяют запустить приложение, обрабатывающее соответствующий документ как единое целое. Автоматизация OLE позволяет управлять этим приложением (сервером OLE), используя экспонированные им функции и процедуры. Осуществляется связь с сервером функцией **CreateOleObject**. Приведем пример, в котором в качестве сервера используется Microsoft Word.

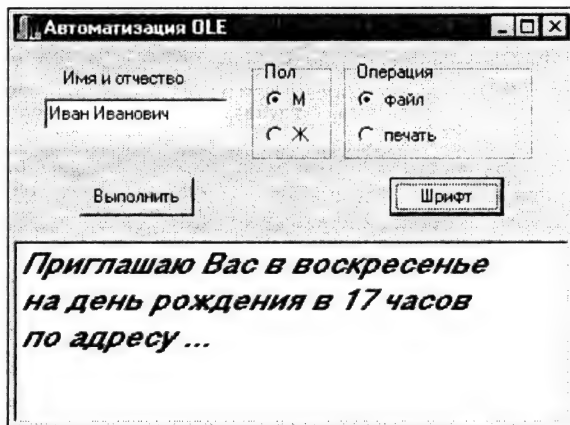
Пусть вы хотите создать приложение, которое печатало бы вам идентичные письма, адресованные различным адресатам. Это могут быть поздравления с праздниками, приглашения на какие-то мероприятия, письма деловым партнерам и т.п. Вы хотите иметь возможность написать в вашем приложении текст письма, выбрать для него атрибуты шрифта, а затем писать имена адресатов и печатать по-

лучившиеся письма или записывать их в файлы для последующей печати или пересылки по электронной почте.

Создайте форму, включающую в себя окно редактирования **Мемо** для ввода текста, окно редактирования **Edit** для ввода имени адресата, группу радиокнопок, определяющих пол адресата, группу радиокнопок, определяющих производимую операцию: запись в файл или печать, диалог **FontDialog** для выбора шрифта, диалог **SaveDialog** для задания имени сохраняемого файла и две кнопки — Шрифт для выбора шрифта и Выполнить для выполнения заданной операции. Пример работы такого приложения приведен на рис. 6.8.

Рис. 6.8

Приложение, использующее автоматизацию OLE



Текст приложения может иметь вид:

```
#include "ComObj.hpp"
Variant W;

...
void __fastcall TForm1::Ins(AnsiString S)
{
    W.Exec(Procedure("Font") << Memol->Font->Name
        << Memol->Font->Size);
    if(Memol->Font->Style.Contains(fsBold))
        W.Exec(Procedure("Bold"));
    if(Memol->Font->Style.Contains(fsItalic))
        W.Exec(Procedure("Italic"));
    W.Exec(Procedure("Insert") << (S + "\n"));
}

//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    AnsiString s;
    W = CreateOleObject("Word.Basic");
    W.Exec(Procedure("AppShow"));
    W.Exec(Procedure("FileNew") << "Normal");
    if(RGSex->ItemIndex == 0)
        Ins("Дорогой " + EName->Text + "!");
    else Ins("Дорогая " + EName->Text + "!");
    for (int i = 0; i <= Memol->Lines->Count; i++)
        Ins(Memol->Lines->Strings[i]);
    if(RGOp->ItemIndex == 0)
        s = "Записать это письмо в файл?";
    else s = "Напечатать это письмо?";
```

```

if (MessageDlg(s, mtConfirmation, TMsgDlgButtons() << mbYes
<< mbNo, 0) == mrYes)
if (RGOp->ItemIndex == 0)
{
if (SaveDialog1->Execute())
W.Exec(Procedure("FileSaveAs") << SaveDialog1->FileName);
}
else W.Exec(Procedure("FilePrint"));
W.Exec(Procedure("FileClose") << 2);
}

//-----
void __fastcall TForm1::BFontClick(TObject *Sender)
{
if (FontDialog1->Execute())
Mem1->Font->Assign(FontDialog1->Font);
}

```

Рассмотрим приведенный код. В приложение необходимо внести директиву препроцессора **#include "ComObj.hpp"**, подключающую файл **ComObj.hpp**, в котором содержится объявление используемой далее процедуры **CreateOleObject**. Вводится переменная **W** типа **Variant**, которая будет являться ссылкой на объект сервера. Основная функция приложения **Button1Click** — обработчик щелчка на кнопку Выполнить. Ее первый выполняемый оператор

```
W = CreateOleObject("Word.Basic");
```

создает ссылку на объект, соответствующий Microsoft Word. При выполнении этого оператора будет запущен Word, если он еще не был запущен к этому моменту. Последующие операторы используют метод **Exec**, который выполняет процедуру или функцию OLE. Как параметр в нее передается конструкция **Procedure(S)**, где **S** — имя вызываемой процедуры WordBasic. Если в процедуру требуется передать параметры, они передаются операций **<<**. Например, во вспомогательной функции **Ins**, заносящей текст в документ Word, вы можете видеть оператор:

```
W.Exec(Procedure("Font") << Mem1->Font->Name
<< Mem1->Font->Size);
```

Этот оператор вызывает процедуру WordBasic **Font**, устанавливающую шрифт, и передает в нее два параметра: имя шрифта и его размер.

Последовательность вызываемых процедур следующая. Сначала вызывается процедура **AppShow**, которая делает окно Word видимым, поскольку по умолчанию запускаемое приложение работает в фоновом режиме. Следующий оператор вызывает процедуру **FileNew**, которая создает новый документ. Затем следуют операторы, формирующие текст документа с учетом установок шрифта. Эти операторы, содержащиеся во вспомогательной функции **Ins**, обращаются к процедурам **Font**, **Bold**, **Italic**, формирующим атрибуты шрифта, и к процедуре **Insert**, включающей текст в документ Word.

После формирования текста следуют операторы, запрашивающие пользователя о завершении операции (это дает ему возможность отказаться от печати или сохранения файла, если он заметит в письме какие-то ошибки). Заключительные операторы завершают заданную операцию процедурами **FileSaveAs** или **FilePrint** и закрывают документ Word процедурой **FileClose** с параметром 2, означающим, что файл закрывается без запроса о необходимости его сохранения.

Запустите приложение, и вы увидите, что научились управлять таким мощным инструментом, как Microsoft Word. Подробное описание других процедур вы сможете найти в справке Word в разделе WordBasic. Одно замечание — если Word открыт до запуска вашего приложения, то позаботьтесь о том, чтобы он не был развернут на весь экран. Иначе окно Word закроет ваше приложение и его сообщения.

6.4.4 Компоненты — серверы COM в C++Builder 5

6.4.4.1 Компоненты — серверы в C++Builder 5

Взаимодействие с Word, Excel и многими другими распространенными программами, входящими в стандартную установку Word и Microsoft Office, может осуществляться из приложений C++Builder 5 с помощью компонентов, размещенных в библиотеке на странице Servers. Эти компоненты отображают множество импортируемых серверов COM. Все они являются потомками своего базового класса **TOLEServer**. В этом классе объявлены абстрактные методы и свойства, позволяющие устанавливать связь с сервером. Поэтому объекты класса **TOLEServer** нельзя создавать непосредственно. В приложениях используются только потомки этого класса — конкретные серверы COM. Они создаются импортом библиотек типов, осуществляемым в среде C++Builder командой `Project | Import`.

Серверы COM — компоненты страницы Servers слабо документированы во встроенной справке C++Builder. Хотелось бы хотя бы частично восполнить этот пробел. Но поскольку в рамках данной книги невозможно рассмотреть подробно все многочисленные серверы страницы Servers, ограничимся только серверами, обеспечивающими связь с Word.

Откройте новое приложение, перенесите на форму компонент **WordApplication** и посмотрите в Инспекторе Объектов его свойства. Их очень немного. Кроме обычных для всех компонентов **Name** и **Tag** имеется всего 4 свойства (во многих компонентах — серверах их всего 3).

Свойство **AutoConnect** определяет, должен ли сервер автоматически загружаться с началом выполнения приложения. Если установить **AutoConnect = true**, то соединение с сервером произойдет в момент начала выполнения вашего приложения. Если же оставить значение **AutoConnect = false**, принятое по умолчанию, то соединение с сервером можно установить вызовом метода **Connect**. Например:

```
WordApplication1->Connect();
```

Впрочем, совершенно не обязательно устанавливать соединение свойством **AutoConnect** или методом **Connect**. Соединение автоматически устанавливается, когда выполняется вызов какого-то метода сервера или задается значение какого-то его свойству.

При использовании свойства **AutoConnect** надо иметь в виду, что установка в **true** влияет только при запуске приложения, т.е. если это свойство установлено во время проектирования. Задание **AutoConnect = true** во время выполнения приложения ни на что не влияет.

Свойство **ConnectKind** определяет, как именно осуществляется соединение с сервером. Это свойство может принимать следующие значения:

ckRunningOrNew	Подсоединиться к выполняющемуся серверу или создать новый экземпляр сервера
ckNewInstance	Всегда создавать новый экземпляр сервера
ckRunningInstance	Только подсоединиться к выполняющемуся серверу
ckRemote	Подсоединиться к удаленному серверу. Эта опция должна сочетаться с заданием свойства RemoteMachineName
ckAttachToInterface	Не подсоединяться к серверу. Вместо этого приложение обеспечивает интерфейс методом ConnectTo (об этом методе будет сказано позднее). Опция ckAttachToInterface не может использоваться совместно с установкой в true свойства AutoConnect

По умолчанию значение **ConnectKind** равно **ckRunningOrNew**. При этом если в момент соединения имеется выполняющийся сервер (применительно к **WordApplication** — если Word открыт), то приложение соединится именно с этим выполняющимся экземпляром сервера. Если же в этот момент соответствующий сервер не выполняется, то будет создан новый экземпляр сервера (в нашем случае будет осуществлен запуск Word).

Если значение **ConnectKind** равно **ckRunningOrNew**, то приложение всегда создает новый экземпляр сервера. Если значение **ConnectKind** равно **ckRunningInstance**, то приложение всегда соединяется с выполняющимся сервером. В этом случае, если выполняющегося сервера нет, будет сгенерировано исключение **EOleSysError**. Поэтому при **ConnectKind = ckRunningInstance**, если нет уверенности, что в момент соединения на компьютере имеется выполняющийся сервер, соединение надо осуществлять, например, так:

```
try
{
    WordApplication1->Connect();
}
catch (EOleSysError&)
{
    Application->MessageBox(
        "На компьютере нет выполняющегося в данный момент Word",
        "Приложение будет закрыто",
        MB_OK + MB_ICONEXCLAMATION);
    Application->Terminate();
}
```

Приведенный код перехватывает исключение **EOleSysError** и выдает пользователю соответствующее сообщение.

Значение **ConnectKind = ckRemote** используется, если надо связаться с удаленным сервером. В этом случае свойство **RemoteMachineName** должно указывать компьютер, на котором выполняется удаленный сервер.

Таким образом, итог рассмотрения свойств, обеспечивающих подключение к серверу, можно подвести следующим образом:

- Если вам надо, чтобы ваше приложение работало с каким-то открытым документом сервера, выполняющимся в момент запуска приложения, следует задать **ConnectKind = ckRunningInstance**
- Если, наоборот, вам надо, чтобы ваше приложение не испортило случайно какой-то документ в уже выполняющемся сервере, следует задать **ConnectKind = ckNewInstance**
- Если вам необходимо связаться с удаленным сервером, надо задать **ConnectKind = ckNewInstance** и установить соответствующее значение **RemoteMachineName**
- В остальных случаях, вероятно, целесообразно сохранять заданное по умолчанию значение **ckRunningOrNew**

Мы не рассмотрели пока значение **ConnectKind** равное **ckAttachToInterface**. Для таких серверов, как **WordApplication**, оно не применимо. Его мы рассмотрим несколько позднее.

После того, как вы установили соединение с сервером, он еще не становится видимым пользователю. Впрочем, приложение может работать с этим сервером, вызывать любые его методы, изменять или читать свойства, но сам сервер останется для пользователю за кадром. Если это нежелательно, если требуется, чтобы пользователь видел, что происходит на сервере, или мог бы сам переключиться на сервер и что-то там сделать, то надо задать свойству **Visible** сервера значение **true**. Например:

```
WordApplication1->Visible = true;
```


Разрыв соединения с сервером осуществляется методом **Disconnect**. Кроме того у таких компонентов COM серверов, как **WordApplication**, имеется свойство **AutoQuit**. Если установить это свойство в **true**, то при завершении приложения автоматически вызовется метод, завершающий сервер. Это свойство полезно устанавливать в **true**, если вы работаете с новым экземпляром сервера, который создало ваше приложение, и после завершения приложения не требуется, чтобы пользователь мог продолжать работу с сервером. Тогда можно или установить **AutoQuit** в **true**, или при завершении приложения выполнить оператор вида:

```
WordApplication1->Disconnect();
```

Это особенно необходимо, если в процессе работы вашего приложения с сервером пользователь не сделал сервер видимым. Если вы не закроете соединение, то после завершения вашего приложения сервер будет по-прежнему невидим, не будет отображен в полосе задач, но в действительности будет выполняться. И когда пользователь решит закрыть Windows, он неожиданно может увидеть какое-то сообщение вашего сервера — невидимки, например, запрос о сохранении файла.

6.4.4.2 Свойства и методы сервера Word

Теперь очень коротко посмотрим, что собой представляет сервер. Любой сервер COM — это объект, имеющий множество свойств, методов, и реагирующий на какие-то события. В этом отношении он ничем не отличается от любого объекта C++Builder. Многие из свойств сервера COM в свою очередь являются объектами со своими методами и свойствами. Описание сервера COM, как правило, можно найти в его встроенной справке. Например, чтобы разобраться в Word как в объекте, надо посмотреть встроенную в него справку. Для этого следует выполнить в Word команду ? | Вызов справки и в открывшейся справке на странице Содержание выбрать раздел Справка по Visual Basic (для версий, младше Word 97, раздел назван WordBasic Reference). Изложение в справке ведется на основе языка Visual Basic, а для младших версий Word — на подмножестве этого языка WordBasic. Конечно, рассмотреть в рамках данной книги даже основы описания Word как объекта невозможно. Поэтому ниже изложены только некоторые начальные сведения, впрочем, достаточные для разработки многих приложений, обращающихся к Word.

Обращение к свойствам объекта **WordApplication**, инкапсулирующего объект **Application** (этот объект является самым выполняющимся экземпляром Word), производится так же, как к свойствам любого объекта C++Builder. Например, в **Application** имеется свойство **Options** — опции, являющееся в свою очередь объектом со множеством свойств. Среди этих свойств есть **CheckSpellingAsYouType** и **CheckGrammarAsYouType** — булевы свойства, указывающие, должен ли Word автоматически проверять синтаксис и грамматику и отмечать в тексте ошибки. Такая проверка замедляет работу Word. Если вы хотите отключить в сервере эти автоматические проверки, введите в приложение операторы:

```
WordApplication1->Options->CheckSpellingAsYouType = false;  
WordApplication1->Options->CheckGrammarAsYouType = false;
```

Тем самым вы отключите автоматические проверки, тем более, что в случае, если Word невидим и работает «за кадром», то эти проверки совершенно бессмысленны.

Среди множества свойств **Application** следует отметить свойство **ActiveDocument** — активный документ. Это объект **Document**, некоторые свойства и методы которого будут описаны ниже.

Практически всегда при работе с сервером Word вам приходится иметь дело со свойством **Documents**. Это свойство представляет собой собрание всех документов, открытых в Word в данный момент. Каждый документ представлен в этом собрании как объект **Document**, имеющий в свою очередь собственные свойства и методы. Общее число открытых документов определяется свойством **Count** собрания

документов **Documents**. Это свойство только для чтения часто приходится проверять, чтобы узнать, есть ли в Word хотя бы один открытый документ. Например, если в вашем приложении предусмотрены действия **ASave**, **APrint** и **APreview**, обеспечивающие сохранение, печать и предварительный просмотр документа, то их, очевидно, надо делать недоступными, если ни одного документа в Word нет. Это можно осуществить следующим кодом:

```
if (WordApplication1->Documents->Count == 0)
{
    ASave->Enabled = false;
    APreview->Enabled = false;
    APrint->Enabled = false;
}
```

Создание нового документа **Document** и включение его в **Documents** осуществляется методом **Add** объекта **Documents**. В этот метод можно передать два аргумента: **Template** и **NewTemplate**. Аргумент **Template** указывает шаблон, который используется при создании документа. Если этот аргумент не указан, то документ создается на основе шаблона Обычный (Normal). Аргумент **NewTemplate** булева типа определяет, открывается ли документ как шаблон (при значении **true**), или как обычный документ. По умолчанию **NewTemplate** = **false**, т.е. отрывается обычный документ.

При вызове из C++Builder любого метода сервера COM аргументы (кроме аргументов типа **Text**) передаются только как объекты типа **OleVariant**. Если какие-то аргументы не являются обязательными, то все равно они должны фигурировать в вызове метода. Только вместо их значений может быть указана **EmptyParam** — переменная типа **OleVariant**, используемая вместо необязательных параметров. Эта переменная объявлена в модуле **System**.

Таким образом, если вы хотите создать новый документ на основе обычного шаблона, вы можете записать оператор:

```
WordApplication1->Documents->Add(EmptyParam, EmptyParam);
```

Но если вы хотите создать документ на основе своего шаблона C:\MyTemplate\My.dot, то код будет сложнее:

```
TVariant Template = "C:\\MyTemplate\\My.dot";
WordApplication1->Documents->Add(&Template, EmptyParam);
```

А если требуется создать документ как новый шаблон на основе обычного шаблона, то код будет таким:

```
TVariant Template = true;
WordApplication1->Documents->Add(EmptyParam, &Template);
```

При передаче булевых аргументов можно использовать значение 0 вместо **false** и целое ненулевое значение (например, 1) вместо **true**. Поэтому, в последнем варианте кода задание значения **NewTemplate** можно выполнить следующим оператором:

```
TVariant Template = 1;
```

Таким образом, при передаче булевых аргументов и свойств можно использовать две различные формы записи значения.

Важным свойством сервера Word является свойство **Selection**, являющееся ссылкой на объект **Selection** — выделенный фрагмент текста в активном документе или, если нет выделения, то просто текущая позиция курсора в активном документе. Этот объект имеет методы **InsertBefore** и **InsertAfter**, аргументом в которые передается текст, вставляемый в активный документ соответственно до или после объекта **Selection**. Например, код:

```
WordApplication1->Selection->InsertAfter(TVariant('\\n'));
WordApplication1->Selection->InsertAfter(
    TVariant("Допорой " + Edit1->Text + "\\n"));
```

вставляет после **Selection** пустую строку, а затем вставляет строку с текстом «Дорогой ... !», где вместо многоточия фигурирует текст окна редактирования **Edit1**. Если выделения текста не было, то все это вставляется после текущей позиции курсора. В результате курсор перемещается на первую позицию после вставленного текста, а весь вставленный текст выделяется, т.е. включается в **Selection**.

То, что введенный текст выделяется, можно использовать для его форматирования. Форматирование объекта **Selection** осуществляется рядом его свойств, из которых остановимся только на двух: **Font** — шрифт и **ParagraphFormat** — формат абзаца. Конечно, можно работать с ними через компонент **WordApplication**. Но гораздо удобнее делать это с помощью специальных серверов COM — компонентов **WordFont** и **WordParagraphFormat**. Это серверы соответственно шрифта и формата абзаца. Таких серверов объектов, входящих как свойство в другие объекты, на странице библиотеки **Servers** много. Подключение их к соответствующему объекту удобнее осуществлять с помощью метода **ConnectTo**. Как аргумент этого метода указывается объект, с которым связывается компонент. Например, операторы

```
WordFont1->ConnectTo(WordApplication1->Selection->Font);
WordParagraphFormat1->ConnectTo(
    WordApplication1->Selection->ParagraphFormat);
```

подключают компоненты **WordFont1** и **WordParagraphFormat1** соответственно к шрифту и формату абзаца выделенного текста. После этого свойства и методы соответствующих объектов можно вызывать через данные компоненты.

Объект **Font** имеет, в частности, следующие свойства: **Name** — имя шрифта, **Bold** — жирный, **Italic** — курсив, **Size** — размер, **StrikeThrough** — перечеркнутый, **DoubleStrikeThrough** — перечеркнутый двойной линией, **Underline** — подчеркнутый, **Shadow** — с тенью, **Emboss** — приподнятый, **Engrave** — утопленный, **Hidden** — невидимый, **Subscript** — нижний индекс, **Superscript** — верхний индекс. Свойство **Underline** может принимать следующие значения: **wdUnderlineNone** — отсутствие подчеркивания, **wdUnderlineSingle** и **wdUnderlineDouble** — одинарное и двойное подчеркивание сплошной линией, **wdUnderlineDash**, **wdUnderlineDotDash**, **wdUnderlineDotDotDash**, **wdUnderlineThick**, **wdUnderlineDotted**, **wdUnderlineWords**, **wdUnderlineWavy** — различные типы линии подчеркивания.

Например, операторы

```
WordFont1->Underline = wdUnderlineSingle;
WordFont1->Bold = 1;
```

обеспечивают в выделенном тексте (если **WordFont1** связан с выделением) жирный подчеркнутый шрифт. Последний оператор можно заменить на эквивалентный ему:

```
WordFont1->Bold = true;
```

Из свойств объекта **ParagraphFormat** и компонента **WordParagraphFormat** отметим только одно: **Alignment** — выравнивание. Оно может принимать значения **wdAlignParagraphLeft** — влево, **wdAlignParagraphCenter** — по центру, **wdAlignParagraphRight** — вправо, **wdAlignParagraphJustify** — по ширине. Например, оператор

```
WordParagraphFormat1->Alignment = wdAlignParagraphCenter;
```

выравнивает текст объекта (выделенного фрагмента, если **WordParagraphFormat1** связан с ним) по центру.

Выше рассматривалась вставка текста с помощью методов **InsertBefore** и **InsertAfter**. Имеется еще один метод вставки текста — **TypeText**. В него так же, как и в методы **InsertBefore** и **InsertAfter**, передается один аргумент — текст. Выполнение метода **TypeText** зависит от значения свойства **ReplaceSelection** объекта **Options**. Если оно установлено в **true**, то все выделение заменяется текстом, пере-

данным как аргумент. Если **ReplaceSelection** = **false**, то новый текст вставляется перед выделением. По умолчанию **ReplaceSelection** = **true**. Таким образом операторы

```
WordApplication1->Options->ReplaceSelection = true;  
WordApplication1->Selection->TypeText(TVariant("новый текст"));
```

приведут к тому, что выделенный текст заменится словами «новый текст». Первый из приведенных операторов можно не писать, если есть уверенность, что значение **ReplaceSelection**, равное **true** по умолчанию, не изменено какими-то предыдущими операторами.

Вставку в документ нового текста или изображения можно осуществлять также методом **Paste**, который копирует информацию из буфера обмена **Clipboard**. Например, операторы

```
#include <Clipbrd.hpp>  
...  
Clipboard()->Assign(Image1->Picture);  
WordApplication1->Selection->Paste();
```

осуществляют копирование с помощью буфера обмена в текущую позицию курсора в документе изображения, хранящегося в компоненте **Image1**.

Выполнение метода **Paste** так же, как и метода **TypeText**, зависит от значения **ReplaceSelection**: копируемая из **Clipboard** информация может или заменять выделение, или вводиться перед ним.

Метод **Collapse** свертывает выделение к его начальной или конечной точке, т.е. снимает выделение, перемещая курсор к его началу или концу. Аргумент **Direction** метода **Collapse** определяет, куда перемещается курсор. При значении **Direction** = **wdCollapseStart** курсор перемещается к начальной точке, а при **Direction** = **wdCollapseEnd** — к конечной. По умолчанию **Direction** = **wdCollapseStart**. Таким образом, операторы

```
TVariant Direction = wdCollapseEnd;  
WordApplication1->Selection->Collapse(&Direction);
```

уберут выделение и переместят курсор в позицию, следующую за бывшим выделением.

Теперь остановимся на объекте **Document** — документ, о котором уже говорилось ранее. С этим объектом удобно работать с помощью специального сервера **COM** — компонента **WordDocument**. Это сервер документа, т.е. объекта, содержащегося в объекте **WordApplication**. Он, как и описанные ранее компоненты **WordFont1** и **WordParagraphFormat1**, подключается к соответствующему объекту с помощью метода **ConnectTo**. Например, оператор

```
WordDocument1->ConnectTo(WordApplication1->ActiveDocument);
```

подключает компонент **WordDocument1** к активному документу, открытому в **Word** и указанному свойством **ActiveDocument**, о котором уже говорилось ранее.

Текст в документе, с которым связывается **WordDocument**, разбивается в свою очередь на объекты **Range**. Каждый такой объект соответствует некоторому непрерывному фрагменту текста. Объект **Range** может создаваться специальным методом **Range**, в котором в качестве начала и конца указываются определенные позиции символов или параграфы. Оператор возвращает указатель на созданный объект. Например, операторы

```
TVariant a = 0, b = 10;  
RangePtr MyRange = WordDocument1->Range(&a, &b);
```

создают объект **MyRange**, включающий первые 10 символов документа.

Свойства и методы объектов **Range** в основном совпадают со свойствами и методами объекта выделения **Selection**, которые мы уже рассматривали. К **Range** можно применять методы **InsertBefore**, **InsertAfter**, **TypeText**, **Paste**, **Collapse**.

Объекты **Range** имеют те же свойства **Font** и **ParagraphFormat**, что и **Selection**. Таким образом, оператор

```
MyRange->Font->Bold = true;
```

форматирует жирным шрифтом фрагмент текста, заключенный в объекте **MyRange**. Если **MyRange** был создан приведенными выше операторами, то форматированию подвергаются первые 10 символов.

Объект документа **Document** имеет свойство **Content**, которое является объектом, отражающим весь текст. Это тоже объект типа **Range** со всеми соответствующими свойствами и методами. Поэтому оператор

```
WordDocument1->Content->Font->Bold = 1;
```

обеспечит форматирование всего компонента жирным шрифтом, а оператор

```
WordDocument1->Content->InsertBefore(TVariant("ЗАГОЛОВОК\n"));
```

вставит перед началом текста строку «ЗАГОЛОВОК».

В заключение остановимся коротко еще на одном свойстве сервера Word — **Dialogs**. Это собрание объектов **Dialog**, которые соответствуют встроенным диалогам Word. Доступ к конкретному диалогу осуществляется через выражение вида

```
WordApplication1->Dialogs->Item(WdWordDialog)
```

где константа **WdWordDialog** может принимать одно из предопределенных значений. В приведенной ниже таблице даются некоторые из этих значений и описание диалогов, которым они соответствуют.

wdDialogEditFind	Найти фрагмент текста
wdDialogEditPasteSpecial	Специальная вставка из буфера обмена
wdDialogEditReplace	Заменить фрагмент текста
wdDialogFileFind	Найти файл
wdDialogFileNew	Новый файл
wdDialogFileOpen	Открыть файл
wdDialogFilePageSetup	Параметры страницы
wdDialogFilePrint	Печать файла
wdDialogFilePrintSetup	Установить принтер
wdDialogFileSaveAs	Сохранить файл как
wdDialogFileSummaryInfo	Свойства документа (Статистика)
wdDialogFormatFont	Шрифт
wdDialogFormatParagraph	Абзац
wdDialogInsertDatabase	Вставить базу данных
wdDialogInsertFile	Вставить файл
wdDialogInsertPageNumbers	Вставить номера страниц

Из методов, которые имеют объекты **Dialog**, остановимся только на одном — методе **Show**. Он открывает пользователю соответствующее диалоговое окно и выполняет те команды, которые указал в нем пользователь.

В этот метод передается один не обязательный аргумент **TimeOut** — время в миллисекундах, после которого диалог автоматически закроется. Если в качестве **TimeOut** передать **EmptyParam**, то диалог закрывается только пользователем.

Например, в коде:

```
WordApplication1->Visible = true;
//Открытие файла
if (WordApplication1->Dialogs->Item(
    wdDialogFileOpen)->Show(EmptyParam) == -1)
...
//Сохранение файла активного документа
WordApplication1->Dialogs->Item(wdDialogFileSaveAs)->Show(EmptyParam);
...
//Печать файла активного документа
WordApplication1->Dialogs->Item(wdDialogFilePrint)->Show(EmptyParam);
```

первый оператор делает сервер видимым. Последующие операторы вызывают диалог открытия файла, диалог сохранения файла активного документа и диалог печати активного документа. Поясним назначение первого оператора приведенного кода. Если в момент вызова диалога сервер был невидимый, то после выполнения диалога открытия файла окно Word станет видимым, но в очень урезанном виде — без главного меню и инструментальных панелей. Поэтому, если нет уверенности, что окно Word было видимо перед вызовом диалога, лучше поместить перед операторами вызова диалога оператор, обеспечивающий видимость сервера.

Вызов диалога методом **Show** возвращает целое значение, позволяющее определить, какой кнопкой пользователь закрыл диалог:

-2	Кнопкой Закрыть
-1	Кнопкой ОК
0	Кнопкой Отмена или клавишей Esc
>0	Одной из командных кнопок: 1 — первой, 2 — второй и т.д.

Впрочем, это общее правило, оговоренное в документации, действует не для всех диалогов. Например, в диалоге открытия файла нажатие кнопки Отмена, системной кнопки Закрыть или клавиши Esc возвращает 0, а нажатие кнопки Открыть (т.е. командной кнопки) возвращает -1. Так что лучше для используемого вами диалога установить экспериментальным путем значения, возвращаемые методом **Show** при тех или иных действиях пользователя.

Возвращенное значение можно использовать для какой-то реакции приложения на действия пользователя. Например:

```
if (WordApplication1->Dialogs->Item(wdDialogFileOpen)->
    Show(EmptyParam) != -1)
    Application->MessageBox("Вы не открыли файл",
        "Надо обязательно открыть один из файлов",
        MB_OK + MB_ICONEXCLAMATION);
```

Многие функции, выполняемые диалогами, могут вызываться как методы документа **Document**, если от пользователя не требуется активных действий. Сохранение активного документа в файле с заданным именем можно осуществить методом **SaveAs**, передавая в него как аргумент типа **OleVariant** имя файла с путем к нему. Если путь отсутствует, то файл сохраняется в текущем каталоге. Вызов метода **SaveAs** можно оформить следующим образом:

```
TVariant FileName = "My.doc";
WordDocument1->SaveAs(&FileName);
```

В данном примере файл документа, с которым связан компонент **WordDocument1**, сохраняется в текущем каталоге с именем «My.doc». Причем **WordDocument1** может быть подключен не обязательно к активному документу.

Печать документа без отображения диалога печати может осуществляться методом **PrintOut**:

```
WordDocument1->PrintOut();
```

Предварительный просмотр документа перед печатью осуществляется методом **PrintPreview**:

```
WordDocument1->PrintPreview();
```

Мы рассмотрели только малую часть свойств методов и объектов сервера Word, реализованную в компонентах **WordApplication**, **WordDocument**, **WordFont** и **WordParagraphFormat**. Упомянем только еще один компонент из того же семейства — **WordLetterContent**. Это объект письма, обеспечивающий работу с шаблонами писем.

6.4.4.3 Тестовый пример работы с сервером Word

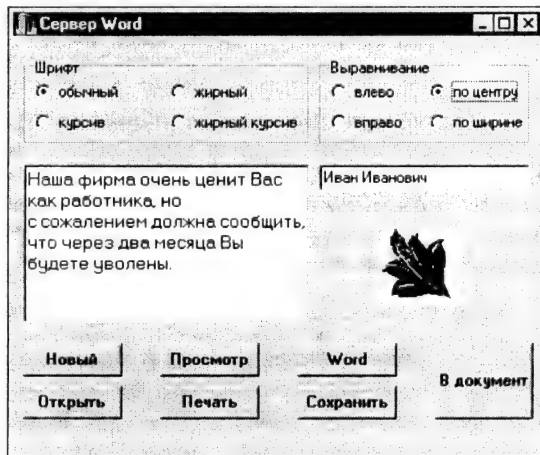
Давайте проверим все, рассказанное в предыдущих разделах, на сравнительно простом примере. Пусть мы хотим построить приложение, напоминающее рассмотренное ранее в разделе 6.4.3. Приложение предназначено для составления серии каких-то писем в формате Word. Введем в приложение возможность управлять шрифтом и выравниванием текста, а также добавим в формируемый документ какой-нибудь рисунок. Все это делается просто, чтобы проверить изложенные выше возможности управления документом Word. Конечно, то же самое можно было бы сделать и в самом Word. Но если вам надо брать данные для писем, например, из какой-то базы данных, то просто одной программой Word обойтись было бы трудно. Позднее в главе 11 в разделе 11.3 будет приведен пример такой работы с сервером Word и базой данных.

Наше тестовое приложение может иметь вид, представленный на рис. 6.9. Соответствующее приложение имеется на прилагаемом к книге диске. Кнопка В документ (ее имя в приведенном ниже тексте приложения **BInDoc**) заносит в активный документ Word текст, пример которого приведен на рис. 6.10. Имя адресата берется из окна редактирования **Edit1**, текст письма — из окна **Memo1**, а рисунок заставки — из компонента **Image1**. Форматирование вводимого текста осуществляется группами радиокнопок Шрифт (ее имя **RGFont**) и Выравнивание (ее имя **RGAlign**).

Кнопка Новый открывает на сервере новый документ. Кнопка Открыть вызывает стандартный диалог Word открытия файла. Кнопка Просмотр вызывает стандартный диалог Word предварительного просмотра документа перед печатью. Кнопка Печать вызывает стандартный диалог печати, а кнопка Сохранить — стандарт-

Рис. 6.9

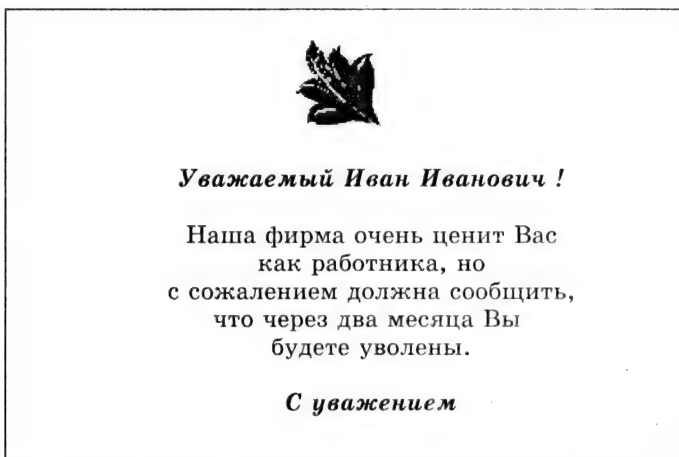
Работа с сервером Word



ный диалог сохранения документа в файле. Кнопка Word делает видимым для пользователя сервер — окно программы Word.

Рис. 6.10

Пример документа,
подготовленного приложением
в сервере Word



Для того, чтобы все это работало, на форме кроме видимых на рис. 6.9 элементов размещены компоненты **WordApplication**, **WordDocument**, **WordFont** и **WordParagraphFormat**. Все свойства этих компонентов равны значениям по умолчанию.

Ниже приводится текст этого приложения.

Заголовочный файл:

```
...
class TForm1 : public TForm
{
__published:      // IDE-managed Components
    TImage *Image1;
    ...
private:           // User declarations
    void __fastcall Connect(void);
    ...
};
```

Файл реализации:

```
...
#include <Clipbrd.hpp>
void __fastcall TForm1::Connect(void)
{
    //Проверка наличия открытого документа
    if (WordApplication1->Documents->Count == 0)
    {
        Application->MessageBox("В Word нет открытого документа",
                                "Команда не выполнена",
                                MB_OK + MB_ICONEXCLAMATION);
        Abort();
    }
    //Соединение WordDocument1 с активным документом
    WordDocument1->ConnectTo(WordApplication1->ActiveDocument);
    //Делаются доступными кнопки работы с документом
    BInDoc->Enabled = true;
    BPreview->Enabled = true;
    BPrint->Enabled = true;
    BSave->Enabled = true;
}
//_____
```

```

void __fastcall TForm1::BNewClick(TObject *Sender)
{
    //Создание в Word нового документа
    WordApplication1->Documents->Add(EmptyParam, EmptyParam);
    Connect();
}
//-----
void __fastcall TForm1::BOpenClick(TObject *Sender)
{
    //Открытие файла
    WordApplication1->Visible = true;
    if (WordApplication1->Dialogs->Item(
        wdDialogFileOpen)->Show(EmptyParam) == -1)
        Connect();
}
//-----
void __fastcall TForm1::BPreviewClick(TObject *Sender)
{
    //Предварительный просмотр
    Connect();
    WordDocument1->PrintPreview();
    WordApplication1->Visible = true;
}
//-----
void __fastcall TForm1::BPrintClick(TObject *Sender)
{
    //Печать
    Connect();
    WordApplication1->Visible = true;
    WordApplication1->Dialogs->Item(wdDialogFilePrint)->Show(EmptyParam);
}
//-----
void __fastcall TForm1::BWordClick(TObject *Sender)
{
    //Word делается ВИДИМЫМ
    WordApplication1->Visible = true;
}
//-----
void __fastcall TForm1::BSaveClick(TObject *Sender)
{
    //Диалог открытия файла
    Connect();
    WordApplication1->Dialogs->Item(
        wdDialogFileSaveAs)->Show(EmptyParam);
}
//-----
void __fastcall TForm1::BInDocClick(TObject *Sender)
{
    //Занесение в документ информации
    //Направление сворачивания выделения
    TVariant Direction = wdCollapseEnd;
    //Пустая строка
    TVariant snw = "\n";
    Connect();
    //Пересылка в документ изображения из Image1
    WordApplication1->Selection->InsertAfter(snw);
    Clipboard()->Assign(Image1->Picture);
    WordApplication1->Selection->Paste();
    WordApplication1->Selection->InsertAfter(snw);
    //Снятие выделения
    WordApplication1->Selection->Collapse(&Direction);
}

```

```

//Включение в документ текста заголовка
WordApplication1->Selection->InsertAfter(snew);
WordApplication1->Selection->InsertAfter(
    TVariant("[>BFBDXfJ " + Edit1->Text + " !\n"));
//Установка жирного курсива и выравнивание по центру
WordFont1->ConnectTo(WordApplication1->Selection->Font);
WordFont1->Bold = 1;
WordFont1->Italic = 1;
WordParagraphFormat1->ConnectTo(
    WordApplication1->Selection->ParagraphFormat);
WordParagraphFormat1->Alignment = wdAlignParagraphCenter;
//Снятие выделения
WordApplication1->Selection->Collapse(&Direction);

//Включение в документ основного текста
WordApplication1->Selection->InsertAfter(snew);
WordApplication1->Selection->InsertAfter(TVariant(Memo1->Text));
WordApplication1->Selection->InsertAfter(snew);
//Установка шрифта и выравнивание основного текста
WordFont1->ConnectTo(WordApplication1->Selection->Font);
WordParagraphFormat1->ConnectTo(
    WordApplication1->Selection->ParagraphFormat);
if (RGFont->ItemIndex > 1)
    WordFont1->Bold = 1;
else WordFont1->Bold = 0;
if (RGFont->ItemIndex % 2 == 1)
    WordFont1->Italic = 1;
else WordFont1->Italic = 0;
switch (RGAlignn->ItemIndex)
{
    case 0: WordParagraphFormat1->Alignment = wdAlignParagraphLeft;
        break;
    case 1: WordParagraphFormat1->Alignment = wdAlignParagraphRight;
        break;
    case 2: WordParagraphFormat1->Alignment = wdAlignParagraphCenter;
        break;
    case 3: WordParagraphFormat1->Alignment = wdAlignParagraphJustify;
}
//Снятие выделения
WordApplication1->Selection->Collapse(&Direction);

//Включение в документ текста заключительной строки
WordApplication1->Selection->InsertAfter(
    TVariant("Y \>BFDRPDX\n\n"));
//Установка жирного курсива и выравнивание по центру
WordFont1->ConnectTo(WordApplication1->Selection->Font);
WordFont1->Bold = 1;
WordFont1->Italic = 1;
WordParagraphFormat1->ConnectTo(
    WordApplication1->Selection->ParagraphFormat);
WordParagraphFormat1->Alignment = wdAlignParagraphCenter;
WordApplication1->Selection->Collapse(&Direction);
}

```

Код содержит, на мой взгляд, достаточно подробные комментарии. Так что ограничимся краткими дополнительными пояснениями. В класс добавлена функция **Connect**. Это вспомогательная функция, вызываемая во многих местах программы. Она проверяет наличие на сервере открытого документа Word и, если имеется открытый документ, то с ним соединяется компонент **WordDocument1**. Правда, в данном приложении компонент **WordDocument** используется только в процедуре **BPreviewClick** для предварительного просмотра документа.

Основная функция кода — **BInDocClick**, обеспечивающая передачу информации в активный документ. Поскольку внутри этой функции изображение, передаваемое в документ, заносится в буфер обмена Clipboard, необходимо инструкцией **#include <Clipbrd.hpp>** подключить модуль **Clipbrd.hpp**, объявляющий функцию **Clipboard**.

В начале функции **BInDocClick** производится обращение к функции **Connect**. Если вызов **Connect** прошел успешно, то далее следует занесение в документ через буфер обмена изображения, хранящегося в **Image1**. После этого методом **Collapse** снимается выделение и в документ заносится требуемый текст из окна **Memo1** с предварительной строкой обращения, в которой используется текст окна **Edit1**. Весь этот текст оказывается выделенным. К этому выделению подключаются компоненты **WordFont1** и **WordParagraphFormat1**, через свойства которых осуществляется форматирование выделенного текста жирным курсивом и выравниванием по центру. Затем выделение снимается, заносится основной текст документа из компонента **Memo1** и он форматируется в соответствии с установками пользователя. После этого выделение с основного текста снимается, заносится заключительная строка и она форматируется жирным курсивом и выравниванием по центру.

Остальные функции приведенного кода вряд ли нуждаются в дополнительных комментариях. Постройте это приложение (или возьмите его с прилагаемого к книге диска) и проверьте в работе. Можете расширить его возможности, чтобы попробовать большинство свойств и методов, рассмотренных в данном разделе.

6.5 Динамический обмен данными — DDE

6.5.1 Общие сведения

Динамический обмен данными (Dinamical Data Exchange — DDE) — это технология, появившаяся в Windows ранее описанного выше OLE, но сохраняющая свое значение и до сих пор, поскольку предоставляет удобный способ обмена данными между программами. В этом диалоге программ одна, инициирующая диалог, называется *клиентом*, а другая, отвечающая на сообщения клиента — *сервером*. Диалог ведется на заданную *тему* (topic). Данные, передаваемые от одной программы к другой, называются *элементами данных* (items). В диалоге могут также передаваться серверу некоторые команды — *макросы* (macros). Этих основных понятий: клиент, сервер, тема, элемент данных и макрос достаточно, чтобы строить приложения, основанные на DDE. Всю сложную задачу обработки в процессе диалога множества сообщений Windows берет на себя C++Builder, инкапсулируя все необходимые процедуры в четырех компонентах, расположенных в библиотеке на странице System: **DdeClientConv**, **DdeClientItem**, **DdeServerConv** и **DdeServerItem**. Первые два из них организуют работу клиента, а два вторые — работу сервера.

Основным компонентом, организующим взаимодействие, является компонент клиента **DdeClientConv**. Соответствующий ему компонент сервера **DdeServerConv** играет пассивную роль, сообщая только тему, которую может поддерживать сервер. Компоненты **DdeClientItem** и **DdeServerItem** нужны только для обмена информацией о конкретных элементах данных.

Взаимодействие DDE может выполнять следующие операции:

- Передача клиенту данных от сервера по инициативе клиента
- Передача клиенту данных от сервера при их обновлении
- Передача данных от клиента к серверу
- Передача команд (макросов) от клиента к серверу
- Установка клиентом данных на сервере

В последующих разделах будут рассмотрены способы решения каждой из этих задач.

6.5.2 Установление контакта с сервером

Начнем рассмотрение с компонента **DdeServerConv**, который располагается на форме приложения-сервера. Его единственное свойство — имя **Name**. Это имя является именем темы. Его должен знать клиент при установлении контакта с сервером. Некоторые методы компонента **DdeServerConv** мы рассмотрим позднее в следующем разделе, а теперь обратимся к компоненту **DdeClientConv**, который располагается на форме приложения-клиента и обеспечивает установление и разрыв контакта с сервером.

Два свойства компонента **DdeClientConv** — **DdeService** и **DdeTopic** определяют имена сервера и темы, соответствующих намеренному контакту. Имя сервера — это имя приложения-сервера, а имя темы — имя одного из компонентов **DdeServerConv**, расположенных на сервере. В процессе проектирования эти свойства задавать не обязательно, поскольку они могут быть установлены во время выполнения методом **SetLink**. Этот метод описан следующим образом:

```
bool SetLink(const System::AnsiString Service,  
            const System::AnsiString Topic);
```

Параметры метода **Service** и **Topic** устанавливают соответственно значения свойств **DdeService** и **DdeTopic**. Установить значения этих свойств во время выполнения прямым присваиванием невозможно.

Способ установления контакта с сервером во время выполнения зависит от значения свойства **ConnectMode**, определяющего режим соединения. Это свойство может иметь значение **ddeManual** — ручной или **ddeAutomatic** — автоматический. При автоматическом режиме **SetLink** не только задает значения **DdeService** и **DdeTopic**, но и устанавливает сам контакт. При ручном режиме метод **SetLink** после установки значений **DdeService** и **DdeTopic** очищает элементы от прошлой информации, но самого соединения не устанавливает. Контакт с сервером в этом случае должен устанавливаться методом **OpenLink**, не имеющим параметров.

Если сервер в момент попытки установить контакт запущен, то контакт устанавливается описанными выше методами. Если это не удастся, то делается попытка запустить программу, путь и имя которой указаны в свойстве **ServiceApplication**. Если в этом свойстве ничего не указано или запуск не удался, то метод **OpenLink** возвращает **false**.

Закрыть контакт с сервером можно методом **CloseLink**. Этот метод разрывает связь с сервером и очищает значения свойств **DdeService** и **DdeTopic**.

Исходя из сказанного выше, наиболее простой способ установления связи вашего приложения с сервером (если приложение будет обмениваться информацией только с одним сервером) заключается в следующем. Во время проектирования надо установить требуемые значения свойств **DdeService** и **DdeTopic**, а затем задать свойство **ConnectMode** = **ddeAutomatic**. Если сервер в данный момент запущен или он доступен, то с сервером установится связь. Доступность в данном случае означает, что выполняемый файл сервера находится в том же каталоге, в котором находится ваше приложение, или в свойстве **ServiceApplication** указан выполняемый файл сервера с полным путем к нему и с указанием типа файла (расширения). В этом случае свойство **ConnectMode** сможет установиться в **ddeAutomatic**. В противном случае вам будет выдано предупреждение о невозможности контакта.

Если все прошло нормально, то в дальнейшем при выполнении приложения связь с сервером будет устанавливаться автоматически. Если в момент запуска приложения сервер уже запущен, контакт установится сразу. Если же сервер не запущен к моменту запуска вашего приложения, но его выполняемый файл доступен для приложения (в указанном выше смысле), то сервер автоматически запустится и с ним будет установлен контакт. В обоих случаях вам не надо будет заботиться о контакте, т.е. не надо будет применять методы **SetLink** или **OpenLink**.

Если указанные выше условия не выполнены, т.е. сервер не запущен ни заранее, ни в момент запуска вашего приложения, то даже, если он потом будет запущен, перед установлением контакта с ним вам потребуется применить метод **SetLink**. Дело в том, что в этих условиях установленные при проектировании значения свойств **DdeService** и **DdeTopic** стираются, а повторно установить их можно только методом **SetLink**. То же самое надо иметь в виду, если вы в своем приложении закрыли соединение с сервером методом **CloseLink** или установили связь с другим сервером. После этого контакт прервется и его надо восстанавливать методом **SetLink**.

Если свойство **ConnectMode** компонента установлено в **ddeManual**, но в процессе проектирования вы установили свойства **DdeService** и **DdeTopic**, то для установления контакта вам достаточно выполнить метод **OpenLink**. Если же **DdeService** и **DdeTopic** не установлены или в процессе работы приложения вы хотите переключаться на связь с другими серверами, то прежде, чем начать работать с сервером, надо выполнить метод **SetLink**, устанавливающий имена сервера и темы. Если при этом **ConnectMode** = **ddeAutomatic**, то автоматически устанавливается контакт с сервером. Если же **ConnectMode** = **ddeManual**, то для установления контакта после **SetLink** надо применить метод **OpenLink**.

Таким образом правила установления контакта с сервером доступным или уже запущенным к момент запуска приложения можно представить следующей таблицей.

ConnectMode	DdeService и DdeTopic	Контакт
ddeAutomatic	Установлены при проектировании	Автоматический
	Не установлены при проектировании, или изменены, или контакт был прерван	SetLink
ddeManual	Установлены при проектировании	OpenLink
	Не установлены при проектировании, или изменены, или контакт был прерван	SetLink OpenLink

Приведем несколько примеров. Пусть ваше приложение будет обмениваться информацией только с одним заранее известным сервером и пусть вы хотите в приложении иметь кнопку (назовем ее **BExchange**), при щелчке на которой вы хотите выполнить какие-то операторы обмена данными с сервером. Вы должны разместить на своем приложении компонент **DdeClientConv** (назовем его **DdeClientConv1**) и эту кнопку.

Далее наиболее простой путь заключается в следующем. Вы устанавливаете в **DdeClientConv1** требуемые значения свойств **DdeService** и **DdeTopic**, а затем задаете свойство **ConnectMode** равным **ddeAutomatic**. Тогда в обработчике события **OnClick** кнопки **BExchange** вам достаточно просто написать операторы обмена данными с сервером.

Если по каким-то причинам во время проектирования сервер вам не доступен, то, задав при проектировании требуемые значения свойств **DdeService** и **DdeTopic**, вы оставляете свойство **ConnectMode**, равным **ddeManual**. В этом случае, например, в обработчике события формы **OnCreate** вы можете поместить оператор

```
if (! DdeClientConv1->OpenLink())
    ShowMessage("Нет контакта с сервером '" +
        DdeClientConv1->DdeService +
        "' по теме '" + DdeClientConv1->DdeTopic + "'");
```

Этот оператор обеспечит контакт с сервером в момент запуска приложения или соответствующее сообщение об ошибке. Другой путь — поместить в обработчик события **OnClick** кнопки **BExchange** операторы

```
DdeClientConv1->SetLink(sServer, sTopic);  
if ( ! DdeClientConv1->OpenLink()  
    ShowMessage("Нет контакта с сервером '" +  
                DdeClientConv1->DdeService +  
                "' по теме '" + DdeClientConv1->DdeTopic + "'");  
< операторы обмена данными с сервером >
```

В этих операторах предполагается, что параметры **sServer** и **sTopic** содержат соответственно имя сервера и имя темы. Отметим, что в этом случае нельзя ограничиться только вторым оператором **OpenLink**. Дело в том, что в первый раз он сработает нормально. Но при последующих щелчках на кнопке **BExchange** он будет выдавать ошибки, поскольку метод **OpenLink** возвращает **false** не только в случае невозможности связаться с сервером, но и в случаях, когда контакт в данный момент уже установлен. Впрочем, это относится только к возвращаемой величине. Если сообщения об ошибках вас не интересуют, то вы можете существенно сократить приведенный код:

```
DdeClientConv1->OpenLink();  
< операторы обмена данными с сервером >
```

Аналогичное приведенному ранее последовательное обращение к методам **SetLink** и **OpenLink** надо применять и в приложениях, работающих с несколькими разными серверами с помощью одного компонента **DdeClientConv**. Такое же обращение используется и в случае, если вы не устанавливали в процессе проектирования свойства **DdeService** и **DdeTopic**.

Все сказанное ранее относилось к установлению контакта с определенным сервером по определенной теме. Но имеется еще одна возможность — установление контакта с заранее неизвестным сервером, который поместил информацию о себе в буфер обмена **Clipboard**. Приложение-сервер, разработанное с помощью **C++Builder**, может сделать это методом **CopyToClipboard** компонента **DdeServerItem** (об этом методе и компоненте будет подробнее рассказано в следующем разделе). Чтобы установить контакт со стороны клиента с сервером, поместившим информацию в буфер обмена, надо выполнить метод **PasteLink** компонента **DdeClientConv**. Например:

```
if ( ! DdeClientConv1->PasteLink()  
    ShowMessage(  
        "В буфере обмена не обнаружено данных, доступных обработке.");
```

Если в буфере обмена есть соответствующие данные, то метод **PasteLink** занесет имена сервера и темы в свойства **DdeService** и **DdeTopic** и установит контакт с этим сервером. Таким образом, этот метод заменяет рассмотренные ранее методы **SetLink** и **OpenLink**, устанавливая контакт с заранее неизвестным сервером.

6.5.3 Обмен данными между клиентом и сервером

6.5.3.1 Построение приложения-сервера

Теперь перейдем к рассмотрению обмена данными между клиентом и сервером. Данные, содержащиеся на сервере, которые могут быть переданы клиентам, обеспечивает компонент **DdeServerItem**. Основные свойства этого компонента — **Lines** — набор строк, представляющий собой передаваемую информацию, и **ServerConv** — имя компонента **DdeServerConv**, т.е. имя темы, к которой относится данная информация. Если свойство **ServerConv** не установлено, то считается, что имя темы — заголовок (**Caption**) формы приложения-сервера.

Свойство **Text** компонента **DdeServerItem** автоматически устанавливается равным первой строке свойства **Lines**. Но если вы устанавливаете свойство **Text** непосредственно, это значение заносится в первую строку **Lines**, а остальные строки очищаются.

Компонент **DdeServerItem** имеет метод **CopyToClipboard**. При выполнении этого метода в буфер обмена Clipboard заносится содержимое хранимой в компоненте информации в специальном формате, который содержит информацию о контакте: имена сервера и темы.

Прежде, чем переходить к вопросу о передаче информации из сервера к клиенту, давайте построим простые серверы, чтобы мы могли проводить с ними эксперименты. Откройте новое приложение и выполните следующие операции:

1. Перенесите на форму компонент **DdeServerConv** и установите его имя **Name** равным **Topic1**. Это будет названием первой темы.
2. Перенесите на форму компонент **DdeServerItem** и установите его имя **Name** равным **Items1**, а свойство **ServerConv** равным **Topic1**. Этот компонент будет хранить содержание первой темы. В свойстве **Strings** задайте какой-нибудь текст, например:

```
строка 1 темы Topic1 сервера Server
строка 2 темы Topic1 сервера Server
```

Этот текст впоследствии на стороне клиента позволит понять, с каким сервером и по какой теме установлен контакт.

3. Перенесите на форму компонент **Edit** с именем **Edit1** и задайте его свойство **Text** равным тексту первой строки свойства **DdeServerItem->Strings**. Этот компонент в дальнейшем потребуется для того, чтобы видеть, как можно наблюдать со стороны клиента изменения информации на сервере. Для этого в обработчик события **OnChange** компонента **Edit1** вставьте оператор

```
Items1->Text = Edit1->Text;
```

переносящий текст из окна редактирования в компонент **Items1**.

4. Повторите пункты 1 и 2, перенеся на форму еще одну пару компонентов **DdeServerConv** и **DdeServerItem** с именами **Topic2** и **Items2**. Они будут отражать вторую тему сервера. Текст темы можете сделать тем же, что и в компоненте **Items1**, заменив в нем слова «Topic1» на «Topic2».
5. Добавьте на форму кнопку **Button**, в обработчик щелчка на которой вставьте оператор

```
Items1->CopyToClipboard();
```

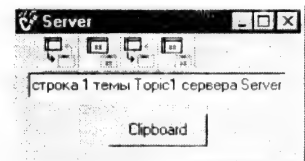
Эта кнопка будет заносить содержание темы в буфер обмена Clipboard.

На этом проектирование сервера закончено. Его форма может иметь вид, представленный на рис. 6.11.

6. Сохраните проект, присвоив ему имя **Server**. Скомпилируйте проект, чтобы создать выполняемый модуль, и проверьте, что все в порядке.
7. Сохраните это проект еще один раз под именем **Server1**. Таким образом, у вас будет два сервера с разными именами, чтобы можно было проверить связь клиента с разными серверами. Замените в текстах компонентов **Items1** и **Items2** этого сервера слова «Server» на «Server1». Создайте выполняемый модуль второго сервера.

Рис. 6.11

Форма приложения-сервера DDE

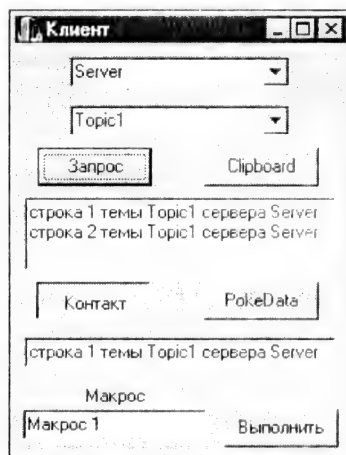


6.5.3.2 Построение приложения-клиента

Теперь рассмотрим приложение-клиент. Общий вид предлагаемого тестового приложения-клиента приведен на рис. 6.12. Но, разместив сразу все его компоненты, код для них мы будем писать постепенно, осваивая различные способы связи с сервером.

Рис. 6.12

Приложение-клиент DDE



Итак, начнем с разработки формы клиента.

1. Перенесите на форму компонент **DdeClientConv** и установите его свойство **ConnectMode** равным **ddeManual**, поскольку мы будем связываться с разными серверами.
2. Перенесите на форму компонент **DdeClientItem** и установите его свойство **DdeConv** равным **DdeClientConv1**.
3. Перенесите на форму два компонента **ComboBox**. Первый из них, предназначенный для выбора сервера, назовите **CBServer**. В его свойство **Items** занесите список серверов: «Server» и «Server1». Второй **ComboBox**, предназначенный для выбора темы, назовите **CBTopics**. В его свойство **Items** занесите список тем: «Topic1» и «Topic2». В обработчик события **OnCreate** формы вставьте операторы, задающие значения индексов обоих компонентов **ComboBox** равными 0.
4. Перенесите на форму компонент **Memo** и два компонента **Edit**. Тексты **Memo** и первого **Edit** (**Edit1**) очистите. Второй компонент **Edit** назовите **EMacro** и задайте в нем какой-нибудь текст, например, «Макрос 1». Этот компонент будет в дальнейшем использоваться для передачи макросов на сервер.
5. Перенесите на форму четыре кнопки **Button**, расположите их согласно рисунку 6.12 и задайте на них надписи "Запрос", "Clipboard", "PokeData" и "Выполнить".
6. Перенесите на форму кнопку **SpeedButton**, назовите ее **SBLink** и установите на ней надпись "Контакт". Эта кнопка должна фиксироваться в нажатом состоянии. Поэтому установите у нее свойство **AllowAllUp** равным **true**, а свойство **GroupIndex** равным 1.

На этом размещение всех компонентов окончено. Теперь можно приступить к рассмотрению различных способов обмена информацией с сервером.

6.5.3.3 Запрос данных сервера

Запросить данные сервера по инициативе клиента можно методом **RequestData** компонента **DdeClientConv**. Этот метод объявлен следующим образом:

```
char * RequestData(const System::AnsiString Item);
```

Здесь **Item** — строка, содержащая имя компонента **DdeServerItem** на сервере, в котором хранится требуемая информация. Функция **RequestData** возвращает эту информацию в виде строки с нулевым символом в конце.

Чтобы воспользоваться этой функцией вы можете в обработчике щелчка на кнопке с надписью **Запрос** написать следующий текст:

```
char Info[256];
if (( ! DdeClientConv1->SetLink(CBServer->Text, CBTopics->Text))
    || ! DdeClientConv1->OpenLink())
    ShowMessage("Нет контакта с сервером '" + CBServer->Text +
               "' по теме '" + CBTopics->Text + "'");
else
{
    Info = DdeClientConv1->RequestData("Items" +
                                      IntToStr(CBTopics->ItemIndex+1));
    Memo1->SetTextBuf(Info);
    DdeClientConv1->CloseLink();
}
```

В начале этого обработчика осуществляется связь методами **SetLink** и **OpenLink** с сервером, заданным пользователем в списке **CBServer** по теме, заданной в списке **CBTopics**. Если связь осуществилась, то в переменную **Info** типа **char** заносится информация от компонента **Items1** или **Items2** на сервере в зависимости от выбранной пользователем темы. Затем эта информация методом **SetTextBuf** заносится в компонент **Memo1**. Последний оператор методом **CloseLink** разрывает связь с сервером.

Отметим, что для осуществления такого запроса в приложении-клиенте вообще не нужен компонент **DdeClientItem**. Он нам потребуется для иных целей. Однако, если уж он имеется на форме, то его тоже можно использовать для получения информации по запросу. Для этого сделайте следующее. В свойстве **DdeItem** компонента **DdeClientItem1** укажите **Items1** — тему по умолчанию. В обработчик события **OnChange** списка **CBTopics** вставьте оператор

```
DdeClientItem1->DdeItem = "Items" + IntToStr(CBTopics->ItemIndex+1);
```

который обеспечит смену темы компонента **DdeClientItem** при изменении этой темы пользователем. В зависимости от выбора пользователя свойству **DdeItem** этот оператор присваивает имя компонента **Items1** или **Items2** на сервере. В обработчик события **OnChange** компонента **DdeClientItem1** вставьте оператор

```
Edit1->Text = DdeClientItem1->Text;
```

Поскольку функция **RequestData** заносит информацию в **DdeClientConv** и в связанный с ним компонент **DdeClientItem**, то при изменении в результате запроса этой информации она отобразится в окне редактирования **Edit1**.

При этом мы продублировали получение информации. В действительности достаточно одной из двух альтернатив:

- Ограничиться приведенной ранее процедурой занесения информации в **Memo1**, и не вводить обработчик события **OnChange** компонента **DdeClientItem**.
- Ввести обработчик события **OnChange** компонента **DdeClientItem**, а в процедуре обработки щелчка на кнопке **Запрос** заменить два оператора чтения информации на один:

```
DdeClientConv1->RequestData("Items" + IntToStr(CBTopics->ItemIndex+1));
```

не используя переменную **Info** и компонент **Memo1**.

Внеся указанные тексты в приложение-клиент, запустите его на выполнение. Предварительно можно запустить (а можно и не запускать) серверы **Server** и **Server1**, с которыми будет осуществляться связь, но не из C++Builder, а, например, программой Windows «Проводник». Запуск из C++Builder одновременно нескольких приложений в режиме отладки возможен только в случае, если вы работаете в Windows NT.

Проверьте наличие связи с разными серверами по разным темам.

Мы рассмотрели запрос к серверу, посылаемый методом **RequestData**. Возможен и другой способ связи с сервером — через буфер обмена Clipboard. Для того, чтобы можно было связаться с сервером, пославшим информацию в Clipboard, существует метод **PasteLink** компонента **DdeClientConv**. Поместите в обработчик щелчка кнопки Clipboard команды

```
DdeClientConv1->PasteLink();
```

Запустите опять ваше приложение на выполнение, запустив одновременно ваши серверы. Нажимая кнопки Clipboard того или иного сервера и затем нажимая кнопку Clipboard клиента, вы увидите, что в данном случае связь осуществляется с тем сервером, который последним поместил информацию в буфер обмена.

6.5.3.4 Постоянное отслеживание информации на сервере

Выше были рассмотрены варианты разовых запросов к серверам. Теперь посмотрим, как можно постоянно наблюдать за сервером, осуществляя его оперативный мониторинг. Сделать это очень просто. Надо соединиться с сервером и, не закрывая этого соединения, установить свойство **DDEItem** компонента **DdeClientItem** равным имени того компонента **DdeServerItem** на сервере, информацию от которого требуется получать. До тех пор, пока соединение открыто, изменяющаяся информация с сервера постоянно будет поступать на компонент **DdeClientItem** клиента.

Чтобы осуществить это в вашем приложении-клиенте, в обработчик щелчка на кнопке **SBLink** с надписью Контакт вставьте следующий код:

```
if (SBLink->Down)
{
    if(( ! DdeClientConv1->SetLink(CBServer->Text, CBTopics->Text))
        || ! DdeClientConv1->OpenLink())
        ShowMessage("Нет контакта с сервером '" + CBServer->Text
            + "' по теме '" + CBTopics->Text + "'");
}
else
{
    DdeClientConv1->CloseLink();
    Edit1->Text = "";
}
```

Если кнопка **SBLink** нажата, то тем же способом, что и ранее, осуществляется связь с сервером. Эта связь разрывается только при отпускании кнопки **SBLink**. Запустите приложение, установите в списках сервер и тему, по которым хотите осуществить связь, и нажмите кнопку Контакт. В окне редактирования появится информация, переданная с сервера. Это все было и ранее. Теперь войдите в приложение-сервер, с которым вы связались, и попробуйте изменять информацию в его окне редактирования. Вы увидите, что все эти изменения немедленно отображаются в окне приложения-клиента. Таким образом между двумя приложениями установлен постоянный контакт. Он прервется только когда вы отпустите в приложении-клиенте кнопку Контакт.

6.5.3.5 Передача информации от клиента к серверу

Все ранее описанное было связано с передачей информации в направлении от сервера к клиенту. Однако имеется возможность передавать информацию и в обратном направлении — от клиента к серверу. Это может осуществляться методами **PokeData**, **PokeDataLines**, **ExecuteMacro** и **ExecuteMacroLines** компонента **DdeClientConv**.

Методы **PokeData** и **PokeDataLines** позволяют изменить информацию, хранящуюся в компоненте **DdeClientItem** на сервере. Их можно использовать, если компонент **DdeClientConv** уже связан с сервером. Методы определены следующим образом:

```
bool PokeData(const System::AnsiString Item, char *Data);
bool PokeDataLines(const System::AnsiString Item,
                   Classes::TStrings* Data);
```

В обеих функциях параметр **Item** определяет имя компонента **DdeServerItem** на сервере, в который заносится информация, а параметр **Data** — тот текст, который должен заменить информацию на сервере. Различие между методами в том, что в **PokeData** в качестве параметра **Data** передается строка с нулевым символом в конце, а в функцию **PokeDataLines** — набор строк типа **TStrings**.

Функции возвращают **true**, если передача данных прошла успешно.

Реализовать подобное изменение информации на сервере в вашем приложении-клиенте можно, например, вставив в обработчик щелчка на кнопке **PokeData** оператор:

```
if ( ! DdeClientConv1->PokeDataLines(
    DdeClientItem1->DdeItem, Memo1->Lines))
    ShowMessage("Данные не переданы, нет контакта с сервером");
```

При записи этого оператора предполагается, что прежде, чем делать щелчок на кнопке **PokeData**, пользователь войдет в контакт с сервером, нажав кнопку **Контакт**. В этом случае в свойстве **DdeClientItem1->DdeItem** хранится имя компонента **DdeServerItem** на сервере, соответствующего теме, которая задана пользователем. В этот компонент **DdeServerItem** передается информация, которую пользователь задал в окне редактирования **Memo1**.

Запустите приложение, установите связь с сервером кнопкой **Контакт**, наберите строки в окне **Memo1** и щелкните на кнопке **PokeData**. Вы увидите, что информация в окне **Edit1** изменится. Это будет свидетельствовать об изменении информации на сервере. Вы можете в этом убедиться, прервав связь с сервером (отжав кнопку **Контакт**) и вновь установив связь кнопкой **Контакт** или **Запрос**.

Выполнение методов **PokeData** и **PokeDataLines** сопровождается событием **OnPokeData** того компонента **DdeServerItem** на сервере, в который передается информация. Вы можете использовать это событие в серверах, чтобы, например, отобразить пользователю полученную информацию. Для этого добавьте в серверы метку **Label** и в обработчик событий **OnPokeData** обоих компонентов **DdeServerItem** внесите оператор

```
Label1->Caption = "Получены данные: '" +
    (TDdeServerItem *)Sender->Text + "'";
```

Этот оператор занесет в надпись метки информацию того компонента **DdeServerItem**, в который она поступила.

Другой способ передачи данных серверу — методы **ExecuteMacro** и **ExecuteMacroLines** компонента **DdeClientConv**. Эти методы не изменяют информацию на сервере, а просто передают на сервер некоторый текст — макрос. Сервер может анализировать переданный макрос и выполнять те или иные действия. Для этого анализа используется событие **OnExecuteMacro** компонента **DdeServerConv** сервера.

Методы **ExecuteMacro** и **ExecuteMacroLines** можно использовать, если компонент **DdeClientConv** уже связан с сервером. Методы определены следующим образом:

```
bool ExecuteMacro(char * Cmd, bool WaitFlg);
bool ExecuteMacroLines(Classes::TStrings* Cmd, bool WaitFlg);
```

Параметр **Cmd** содержит строку макроса (в **ExecuteMacro**) или набор макросов (в **ExecuteMacroLines**), которые должны передаваться на сервер. Параметр **WaitFlg** определяет, должен ли клиент ждать завершения выполнения сервером всех макросов. Если **WaitFlg = true**, то последующие вызовы методов **ExecuteMacro**, **ExecuteMacroLines**, **PokeData**, **PokeDataLines** и **RequestData** не будут успешными, пока сервер не обработает всех переданных ему макросов. Проверить, завершился ли сервер обработку предыдущих макросов, можно, обратившись к свойству **WaitStat** компонента **DdeClientConv**. Пока **WaitStat = true**, вызов методов **ExecuteMacro**, **ExecuteMacroLines**, **PokeData**, **PokeDataLines** и **RequestData** не имеет смысла, так как сервер занят и эти методы все равно не будут выполнены.

Функции возвращают **true** при успешной передаче макросов серверу.

Обработчик события **OnExecuteMacro** компонента **TDdeServerConv**, которому передан макрос, имеет заголовок вида:

```
void ... (TObject *Sender, TStrings *Msg)
```

Параметр **Msg** содержит тексты переданных макросов.

Чтобы ввести передачу макросов в наш тестовый пример, сделайте следующее. В приложении-клиенте в обработчик щелчка на кнопке Выполнить вставьте код:

```
if (!DdeClientConv1->ExecuteMacro(EMacro->Text.c_str(), false))
    ShowMessage("Макрос не выполнен, нет контакта с сервером");
```

Этот оператор передает на сервер в качестве макроса строку, набранную пользователем в окне редактирования **EMacro**.

На серверах надо ввести обработчики событий **OnExecuteMacro** компонентов **TDdeServerConv**. Давайте в нашем тестовом примере просто ограничимся отображением переданного макроса в надписи метки **Label1**. Тогда обработчик события **OnExecuteMacro** обоих компонентов **TDdeServerConv** сервера может иметь вид:

```
void __fastcall TFServer::Topic1ExecuteMacro(TObject *Sender,
                                             TStrings *Msg)
{
    if (Msg->Count == 0 )
        ShowMessage(Application->ExeName + ": Макрос не получен");
    else
        Label1->Caption = "Получен макрос '" + Msg->Strings[0] + "'";
}
```

Теперь запустите вне C++Builder ваши серверы (не обязательно), запустите на выполнение приложение-клиент и проверьте его работу по передаче макросов и во всех остальных режимах.

Глава 7

Повторное использование разработанных кодов

7.1 Способы сохранения и повторного использования кодов

Повторное использование кодов — важнейшая концепция объектно-ориентированного визуального программирования. Все компоненты C++Builder — это коды, разработанные фирмой Borland, которые вы повторно используете, включая их в свое приложение. Вы используете также явно или неявно множество функций и процедур API Windows, применяя многие функции и процедуры языка C++. Но в данном разделе в основном речь пойдет о другом — как вам сохранить и повторно использовать свои собственные коды, разработанные в процессе проектирования.

В C++Builder предусмотрено несколько механизмов сохранения и повторного использования кодов:

- Создание и хранение в библиотеке шаблонов компонентов
- Хранение и наследование форм и фреймов в Депозитарии
- Хранение проектов в Депозитарии
- Создание новых компонентов и хранение их в библиотеке
- Создание динамически присоединяемых библиотек DLL
- Создание пакетов (packages)

Первые четыре механизма позволяют вам включать ранее разработанные коды непосредственно в ваш загрузочный модуль. Два последних подхода дают возможность включать коды в некоторые файлы поддержки, которые должны распространяться совместно с вашим приложением. Как увидим ниже, такой подход имеет свои преимущества и недостатки.

7.2 Создание и хранение шаблонов компонентов

Наиболее простой способ сохранения разработанных вами компонентов для последующего использования — запоминание в библиотеке визуальных компонентов их шаблонов. Покажем, как это делается, на простом примере. Пусть вы хотите на основе компонента **Edit** создать окно редактирования, в котором пользователь мог бы вводить только целые числа, т.е. не мог бы вводить никаких других символов, кроме цифр. Кроме того, при нажатии пользователем клавиши Enter фокус должен передаваться следующему компоненту в последовательности табуляции. Это можно сделать, например, следующим образом.

Откройте новое приложение и перенесите на форму компонент типа **Edit**. В обработчике его события **OnKeyPress** напишите оператор

```
Set <char, '0', '9'> Dig;  
if ( ! (Dig << '0' << '1' << '2' << '3' << '4' << '5'  
      << '6' << '7' << '8' << '9').Contains(Key) )  
    Key = 0;
```

а в обработчике события **OnKeyDown** — оператор:

```
if (Key == VK_RETURN)
    FindNextControl(
        (TwinControl *)Sender, true, true, false)->SetFocus();
```

Эти операторы, о которых подробнее рассказано было в разделе 4.3.2.2, заменяют все символы, кроме цифр, нулевым символом, который не будет отображаться в окне, и при нажатии Enter передадут фокус очередному компоненту.

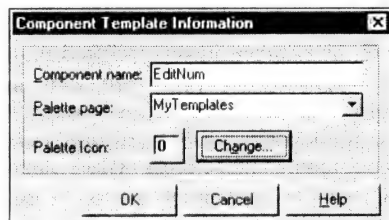
Измените также текст в окне (свойство **Text**) на «0».

Пусть мы хотим сохранить шаблон этого компонента в библиотеке визуальных компонентов, чтобы в дальнейшем включать подобный компонент в другие свои приложения. Для этого надо, выделив предварительно данный компонент, выполнить команду **Component | Create Component Template** (создать шаблон компонента). В открывшемся диалоговом окне **Component Template Information** (рис. 7.1) вы можете задать имя компонента (в верхнем окне редактирования), которое будет появляться на ярлычке подсказки, если пользователь задержит курсор мыши над пиктограммой этого компонента в библиотеке визуальных компонентов. На рис. 7.1 это имя — **EditNum**. В выпадающем списке в средней части окна вы можете выбрать страницу библиотеки визуальных компонентов, на которой вы хотите разместить пиктограмму компонента. Вы можете также указать новое имя (**MyTemplates** на рис. 7.1) и тогда в библиотеке визуальных компонентов будет создана новая страница с этим именем. Можно также изменить пиктограмму данного компонента (кнопка **Change**). Пиктограмма, если вы ее хотите сменить, должна быть подготовлена заранее в виде файла **.bmp** размером 24 на 24. Как это можно сделать, рассказано в разделе 5.1.2.2 главы 5. На рис. 7.1 как раз загружается пиктограмма, создание которой описано в разделе 5.1.2.2. После выполнения всех описанных операций щелкните на кнопке **OK**.

Ваш шаблон появится в библиотеке. Вы можете убедиться в этом, посмотрев на указанную вами страницу библиотеки.

Рис. 7.1

Окно ввода информации о новом шаблоне компонента



Теперь попробуйте создать новое приложение и перенести на форму созданный вами компонент. Вы увидите, что он появится на форме в том виде, в котором вы его записали в библиотеку. Посмотрев на страницу событий этого компонента в Инспекторе Объектов, вы увидите, что на ней указаны обработчики событий **OnKeyDown** и **OnKeyPress**, а в процедурах этих обработчиков уже записаны те операторы, которые вы написали перед созданием шаблона.

Правда, у такого подхода имеется определенный недостаток. Если вы разместите на форме несколько созданных вами компонентов, то для каждого из них в текст модуля включатся соответствующие обработчики событий. Это явная избыточность, поскольку написанные вами операторы обработки универсальны и хватило бы одного обработчика события **OnKeyDown** и одного обработчика события **OnKeyPress** на все размещенные компоненты. Чтобы избежать избыточности, желательно будет оставить в модуле по одному обработчику каждого вида, остальные удалить из текста, а во всех окнах редактирования сослаться на оставшиеся обработчики.

Таким образом вы можете создать себе немало шаблонов компонентов, которые кочуют из приложения в приложение. Аналогичным образом можно создавать

шаблоны не только отдельных компонентов, но и групп компонентов. Например, в разделе 4.1.6.2 описано создание достаточно универсального меню и инструментальной панели. Эту группу компонентов целесообразно сохранить в виде шаблона. Вообще целесообразно сделать шаблоны всех тех фрагментов, которые часто используются вами при проектировании форм. Это сэкономит вам немало времени при разработке новых проектов.

Если в дальнейшем вам перестанет нравиться созданный вами шаблон, вы можете удалить его из библиотеки. Описание этой процедуры вы найдете в разделе 14.2.2 главы 14.

7.3 Создание новых компонентов и включение их в библиотеку

7.3.1 Начало создания и установка компонента

Создание новых компонентов дает, конечно, неизмеримо больше возможностей, чем построение шаблонов имеющихся компонентов. При создании компонентов, наследующих каким-либо имеющимся в C++Builder компонентам, вы можете добавить новые свойства, в том числе и отображаемые в Инспекторе Объектов, добавить новые события и т.п. Однако, создание новых компонентов — это непростой процесс, требующий хорошего понимания тонкостей наследования и хорошего знания свойств, методов и событий родительских компонентов. Подробное рассмотрение этих вопросов выходит за рамки настоящей книги. Тем не менее некоторые основы этого будут рассмотрены ниже.

Давайте поставим перед собой сравнительно простую задачу: создать окно редактирования, в котором по желанию во время проектирования и во время выполнения можно будет разрешать ввод только цифр (например, если предполагается, что пользователь должен вводить целое число), запрещать ввод цифр (например, при вводе пользователем фамилии, имени и отчества) или разрешать ввод любых символов. Кроме того предусмотрим очистку содержимого окна и свойство, показывающее, был ли модифицирован пользователем текст с момента последней очистки. В момент очистки определим генерацию соответствующего события, которое пользователь при желании может обрабатывать.

Построим наш компонент как наследника класса **TEdit** и назовем наш новый класс **TEditLetNum**. В компонент, помимо свойств, обычных для **TEdit**, желательным будет добавить два новых свойства: **EnableNum** и **EnableLet**. Тип обоих свойств — **bool**. Первое из них разрешает или запрещает ввод цифр, а второе разрешает или запрещает ввод каких-либо символов, кроме цифр. Таким образом, если **EnableNum = true** и **EnableLet = true**, то это будет обычное окно редактирования. Если **EnableNum = true**, а **EnableLet = false**, получим окно, в котором можно вводить только цифры. Если **EnableNum = false**, а **EnableLet = true**, то в окно, запрещено вводить цифры. Ну а ситуацию, когда **EnableNum = false** и **EnableLet = false**, надо запретить, поскольку в такое окно ничего ввести невозможно.

Предусмотрим в компоненте метод **Clear** — очистку текста в окне и свойство **Modified**, которое будет показывать, был ли модифицирован текст с момента последней очистки. В момент очистки будем генерировать событие **OnClear**.

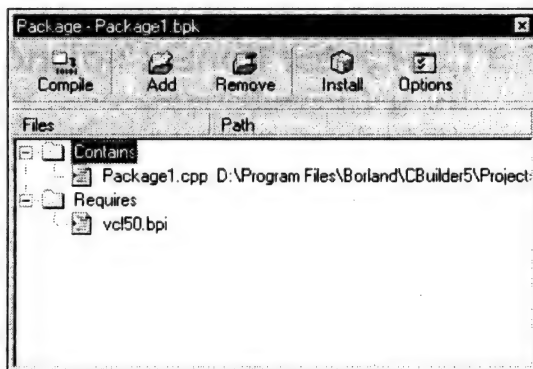
Если вы хотите, чтобы ваш новый компонент фигурировал на страницах палитры компонентов с новой оригинальной пиктограммой, то начните с ее создания. Необходимые для этого действия описаны в разделе 5.1.2.4 главы 5.

Компоненты в C++Builder компилируются в пакеты. Поэтому, если вы еще не создали пакет, в который хотите компилировать ваш новый компонент, то можете начать с его создания (впрочем, это можно сделать и позднее). Для создания нового пакета:

1. Выполните команду **File | New** и в диалоговом окне **New Items** на странице **New** выберите пиктограмму **Package** — пакет. После этого вы попадете в окно Диспетчера Пакетов (**Package Manager**), показанное на рис. 7.2. В этом окне **Contains** — содержимое пакета, а **Requires** — список других пакетов, требующихся для поддержки вашего.
2. Сразу сохраните пакет в файле с расширением **.dpk**, выполнив команду **File | Save As** или выбрав команду **Save** из меню, всплывающего при щелчке правой кнопкой мыши.

Рис. 7.2

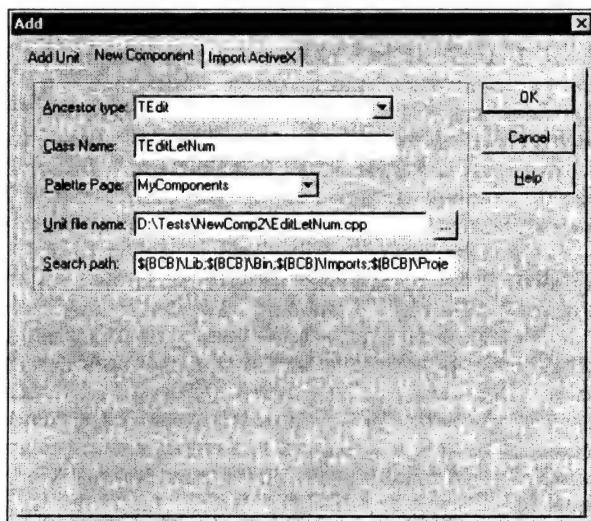
Окно Диспетчера Пакетов, готовое для внесения нового компонента



3. Щелкнув на кнопке **Add** (добавить), вы попадете в окно, позволяющее вам включить в пакет модуль (**Add Unit**), компонент (**New Component**) или **ActiveX** (**Import ActiveX**). В данном случае вам надо включить в пакет свой новый компонент. Для этого вы открываете соответствующую страницу, которая имеет вид, представленный на рис. 7.3. Укажите в этом окне родительский тип (выберите **TEdit** из выпадающего списка **Ancestor type**), имя нового класса в окне **Class Name** (для нашего примера — **TEditLetNum**), страницу в библиотеке компонентов (**Palette Page**), на которой хотите разместить пиктограмму вашего нового компонента. Можете указать новую страницу и она будет создана. Проверьте также путь и имя модуля вновь создаваемого компонента (окно **Unit file name**).

Рис. 7.3

Окно ввода информации о включаемом в пакет компоненте



4. Щелкните на ОК и вы вернетесь в окно Диспетчера Пакетов, но в нем уже появится имя вашего компонента. А если вы заранее создали файл ресурса компонента **.dcr** с его новой пиктограммой, то Диспетчер Пакетов включит в пакет и этот файл.
5. Щелкните на кнопке **Install** (установка) и компонент будет установлен на указанной вами странице библиотеки компонентов. Можете открыть эту страницу и увидеть его там. Выполнив в окне Диспетчера Пакетов двойной щелчок на имени файла компонента, вы можете перейти в окно редактирования и увидеть, что в нем появилась заготовка модуля вашего компонента. Сохраните ее в файле (**File | Save**).
6. Чтобы скомпилировать модуль вашего компонента, можно щелкнуть в окне Диспетчера Пакетов на кнопке **Compile** (компиляция). Но в данном случае модуль уже скомпилирован во время его установки.

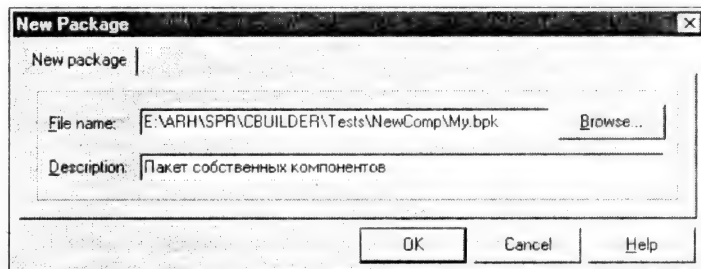
При выходе из Диспетчера Пакетов вам будет задан вопрос о сохранении информации в файле пакета (расширение этого файла **.dpk**). Ответьте на вопрос утвердительно.

Возможно и другое начало проектирования нового компонента.

7. Выполните команду **Component | New Component** и попадете в почти такое же, как описано выше, окно **New Component**.
8. Занеся в него необходимую информацию, щелкните на кнопке **Install** (установка) и перед вами появится диалоговое окно (рис. 7.4) с двумя закладками: **Into existing package** (устанавливать в существующий пакет) и **Into new package** (устанавливать в новый пакет). Выберите файл пакета из существующих или, лучше, укажите имя нового файла и строку его описания на странице **Into new package**. Вам будет задан вопрос: «File ... will be built then installed. Continue?» («Файл ... будет сначала построен, а потом установлен. Продолжать?»). Ответьте на заданный вопрос утвердительно и файл будет установлен в пакет, зарегистрирован на указанной странице в библиотеке, а его модуль появится в окне редактирования. На экране появится рассмотренное ранее окно Диспетчера пакетов (рис. 7.2).

Рис. 7.4

Окно ввода информации о новом пакете



Мы рассмотрели пути создания новых пакетов и включения в них новых компонентов. Если в дальнейшем вы захотите удалить зарегистрированный вами пакет, это можно сделать одним из двух способов: Выполнить команду **Component | Install Packages** или команду **Project | Options**. В первом случае вы сразу попадете в диалог работы с пакетами. Во втором случае этот диалог вы найдете на странице **Packages**. Диалог работы с пакетами будет подробно рассмотрен в разделе 7.6.2 (рис. 7.16). В этом диалоге вы можете удалить ваш пакет кнопкой **Remove**. Впрочем, если вы в дальнейшем передумаете, вы опять можете зарегистрировать его кнопкой **Add**.

Итак, вы создали прообраз вашего компонента и зарегистрировали его. По умолчанию пиктограмма компонента на соответствующей странице библиотеки

будет взята тождественной пиктограмме родительского типа. А если вы заранее, как указывалось в разделе 5.1.2.4 главы 5, создали файл ресурсов компонента и включили в него новую пиктограмму, то именно ею вы сможете любоваться в палитре компонентов на указанной вами странице.

7.3.2 Структура класса компонента

Заготовка модуля компонента, созданная в результате указанных ранее действий, имеет следующий вид.

Файл EditLetNum.h:

```
#ifndef EditLetNumH
#define EditLetNumH
//_____
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
#include <StdCtrls.hpp>
//_____
class PACKAGE TEditLetNum : public TEdit
{
private:
protected:
public:
    __fastcall TEditLetNum(TComponent* Owner);
    __published:
};
//_____
#endif
```

Файл EditLetNum.cpp:

```
#include <vcl.h>
#pragma hdrstop

#include «EditLetNum.h»
#pragma package(smart_init)
//_____
// ValidCtrCheck is used to assure
// that the components created do not have
// any pure virtual functions.

static inline void ValidCtrCheck(TEditLetNum *)
{
    new TEditLetNum(NULL);
}
//_____
__fastcall TEditLetNum::TEditLetNum(TComponent* Owner)
    : TEdit(Owner)
{
}
//_____
namespace Editletnum
{
    void __fastcall PACKAGE Register()
    {
        TComponentClass classes[1] = {__classid(TEditLetNum)};
        RegisterComponents(«MyComponents», classes, 0);
    }
}
```

Собственно говоря, в этой заготовке пока только каркас класса будущего модуля. Файл **EditLetNum.cpp** содержит три процедуры: **ValidCtrCheck**, конструктор **TEditLetNum** и **Register**. Процедура **ValidCtrCheck** носит вспомогательный характер и вводится, чтобы проверять, не содержит ли компонент чистых виртуальных функций. Тело конструктора **TEditLetNum** пока пустое. Позднее мы заполним его. А процедуру **Register**, регистрирующую компонент на заданной странице библиотеки, давайте рассмотрим подробнее.

Код регистрации компонента начинается с оператора **namespace**. Ключевое слово **namespace** устанавливает локальность имен данной процедуры регистрации. После этого ключевого слова следует имя файла, содержащего компоненты. Имя пишется символами в нижнем регистре, кроме первой заглавной буквы.

В процедуре регистрации **Register** первый оператор создает массив регистрируемых компонентов **classes** типа **TComponentClass** и заносит в него регистрируемый компонент. Если бы вы создали два компонента (пусть имя второго из них **TEdit2**), регистрируемых на одной странице библиотеки, вы могли бы занести их в массив оператором:

```
TComponentClass classes[2] = { __classid(TEditLetNum),
                               __classid(TEdit2) };
```

Следующий оператор процедуры регистрации регистрирует функцией **RegisterComponents** компоненты, занесенные в **classes** (второй параметр функции) на странице **MyComponents** (первый параметр). Последний параметр является последним индексом массива регистрируемых компонентов.

Теперь рассмотрим коротко описание класса в заголовочном файле **EditLetNum.h**. Вы видите в нем разделы **private**, **protected**, **public** и **__published**. Они определяют четыре варианта доступа к переменным, процедурам и функциям:

private (закрытые)	Процедуры и функции, определенные таким образом, доступны только в пределах данного модуля
protected (защищенные)	Процедуры и функции, определенные таким образом, доступны в классах потомков
public (открытые)	Эти процедуры и функции доступны везде
__published (опубликованные)	Процедуры и функции доступны везде и имеют связь со средой разработки C++Builder, обеспечивающую вывод на экран в Инспекторе Объектов страниц информации о свойствах и событиях

Вспомним (см. раздел 1.2 главы 1) наше определение объекта (компонент — это объект) как совокупности свойств, методов и обработчиков событий. Причем свойства — это совокупность полей данных и методов их чтения и записи. Вспомним также принцип скрытия информации. Исходя из этого, разработчик компонента должен продумать, в какие разделы вставить вводимые им поля данных, свойства и методы.

7.3.3 Задание свойств

Поля данных всегда должны быть защищены от несанкционированного доступа. Поэтому их целесообразно определять в **private** — закрытом разделе класса. В редких случаях их можно помещать в **protected** — защищенном разделе класса, чтобы возможные потомки данного класса имели к ним доступ. Традиционно идентификаторы полей совпадают с именами соответствующих свойств, но с до-

бавлением в качестве префикса символа 'F'. Таким образом, в нашем примере вы можете занести в раздел **private** объявления трех необходимых нам полей данных:

```
class PACKAGE TEditLetNum : public TEdit
{
private:
// Закрытые элементы-данные класса
    bool FEnableNum;
    bool FEnableLet;
    bool FModified;
    ...
};
```

Теперь надо объявить свойства — методы чтения и записи этих полей. Свойство объявляется оператором вида:

```
__property <тип> <имя> = {read=<имя поля или метода чтения>
                           write=<имя поля или метода записи>
                           <директивы запоминания
                           и значения по умолчанию>;
```

Если в разделах **read** или **write** записано имя поля, значит предполагается прямое чтение или запись данных.

Если в разделе **read** записано имя метода чтения, то чтение будет осуществляться только функцией с этим именем. Функция чтения — это функция без параметра, возвращающая значение того типа, который объявлен для свойства. Имя функции чтения принято начинать с префикса **Get**, после которого следует имя свойства.

Если в разделе **write** записано имя метода записи, то запись будет осуществляться только процедурой с этим именем. Процедура записи — это процедура с одним параметром того типа, который объявлен для свойства. Имя процедуры записи принято начинать с префикса **Set**, после которого следует имя свойства.

Если раздел **write** отсутствует в объявлении свойства, значит это свойство только для чтения и пользователь не может задавать его значение.

Директивы запоминания определяют, как надо сохранять значения свойств при сохранении пользователем файла формы **.dfm**. Чаще всего используется директива

```
default = <значение по умолчанию>
```

Она не задает начальные условия. Это дело конструктора. Директива просто говорит, что если пользователь в процессе проектирования не изменил значение свойства по умолчанию, то сохранять значение свойства не надо.

Итак, для нашего примера объявления свойств могут иметь вид:

```
public:
    __fastcall TEditLetNum(TComponent* Owner);
// Свойство только времени выполнения
    __property bool Modified = {read=FModified, default=false};

    published:
// Свойства компонента, включаемые в Инспектор Объектов
    __property bool EnableLet = {read=FEnableLet,
                                write=SetEnableLet,
                                default=true};
    __property bool EnableNum = {read=FEnableNum,
                                write=SetEnableNum,
                                default=true};
```

Объявление свойства **Modified** помещается в раздел **public**, поскольку это свойство должно быть доступно только во время выполнения. Свойства

EnableNum и **EnableLet** помещаются в раздел **__published**, так как они должны отображаться в Инспекторе Объектов во время проектирования.

Свойства **EnableNum** и **EnableLet** имеют прямой доступ к полям для чтения. Но для записи они имеют методы **SetEnableNum** и **SetEnableLet** соответственно. Это связано с тем, что при записи свойств надо проверять, не окажутся ли значения обоих этих свойств равными **false**. Подобное задание значений надо предотвращать, поскольку в этом случае в окно редактирования вообще ничего нельзя будет ввести.

Свойство **Modified** вообще не имеет метода записи, поскольку оно предназначено только для чтения.

Указанные в объявлениях методы записи могут быть реализованы обычными функциями, объявления которых помещаются в **private** — закрытый раздел класса, а их реализация включается в тело модуля. Для того, чтобы задать свойствам начальные значения, надо еще включить соответствующие операторы в тело конструктора. В итоге в данный момент модуль компонента может иметь следующий вид.

Файл **EditLetNum.h**:

```
...
class PACKAGE TEditLetNum : public TEdit
{
private:
    // Закрытые элементы-данные класса
    bool FEnableLet;
    bool FEnableNum;
    bool FModified;
protected:
    // Защищенные методы записи
    void __fastcall SetEnableLet(bool AEnableLet);
    void __fastcall SetEnableNum(bool AEnableNum);
public:
    // Объявление конструктора
    __fastcall TEditLetNum(TComponent* Owner);
    // Свойство только времени выполнения
    __property bool Modified = {read=FModified, default=false};
    __published:
    // Свойства компонента, включаемые в Инспектор Объектов
    __property bool EnableLet = {read=FEnableLet,
                                write=SetEnableLet,
                                default=true};
    __property bool EnableNum = {read=FEnableNum,
                                write=SetEnableNum,
                                default=true};
};
...
```

Файл **EditLetNum.cpp**:

```
...
static inline void ValidCtrCheck(TEditLetNum *)
{
    new TEditLetNum(NULL);
}
// _____
__fastcall TEditLetNum::TEditLetNum(TComponent* Owner)
    : TEdit(Owner)
{
    FEnableLet = true;
    FEnableNum = true;
    FModified = false;
}
```

```

//-----
namespace Editletnum
{
    void __fastcall PACKAGE Register()
    {
        TComponentClass classes[1] = {__classid(TEditLetNum)};
        RegisterComponents(«MyComponents», classes, 0);
    }
}
//-----
void __fastcall TEditLetNum::SetEnableNum(bool AEnableNum)
{
    //Присваивание значения полю FEnableNum
    FEnableNum = AEnableNum;

    // Если значения FEnableNum и FEnableLet = false,
    // то полю FEnableLet присваивается true
    if (! AEnableNum)
        if (! FEnableLet) FEnableLet = true;
}
//-----
void __fastcall TEditLetNum::SetEnableLet(bool AEnableLet)
{
    //Присваивание значения полю FEnableLet
    FEnableLet = AEnableLet;

    // Если значения FEnableNum и FEnableLet = false,
    // то полю FEnableNum присваивается true
    if (! AEnableLet)
        if (! FEnableNum) FEnableNum = true;
}

```

Процедуры, реализующие методы записи, присваивают полю переданное в них значение параметра и в случае, если передано значение **false**, проверяют значение другого поля. Если и другое поле имеет значение **false**, то оно исправляется на **true**.

Объявления процедур записи включены в раздел **protected**. Это означает, что они закрыты для внешнего пользователя, но доступны для возможных потомков вашего класса.

Конструктор объявлен в открытом разделе класса **public** и имеет имя **TEditLetNum**, совпадающее с именем класса. В реализации конструктора задаются начальные значения новым свойствам, которые вы добавили в компонент.

Теперь давайте сохраним подготовленные файлы, откомпилируем их (кнопка **Compile** в Диспетчера Пакетов рис. 7.2) и построим тестовое приложение для отладки установленного компонента. Хотя он еще не до конца создан, кое-что уже можно увидеть.

Откройте новый проект и внесите в него с соответствующей страницы ваш новый компонент. Откройте (командой **File | Open**) файл вашего компонента, чтобы можно было вносить в него какие-то изменения. Сохраните проект под каким-нибудь именем (например, **Test**). Откомпилируйте ваше приложение (команда **Project | Make All Projects**). Если все прошло без ошибок, выделите на форме ваш компонент и взгляните на его свойства в Инспекторе объектов. Вы найдете там добавленные вами свойства **EnableNum** и **EnableLet**. Оба свойства имеют по умолчанию заданные в конструкторе значения **true**. Попробуйте изменить оба значения на **false**. Это вам не удастся сделать, так как сработают написанные вами методы записи.

Свойство **Modified** в Инспекторе объектов вы не увидите, так как это свойство может использоваться только во время выполнения.

7.3.4 Создание методов

Методы включаются в открытый раздел класса — раздел **public**, поскольку иначе их нельзя было бы использовать. Собственно один метод — конструктор **TEditLetNum**, вы уже написали. Теперь вам нужно написать метод **Clear**, очищающий содержимое окна редактирования и устанавливающий значение поля **FModified** равным **false**. Генерировать событие **OnClear** мы пока не будем. А поскольку базовый класс **TEdit** уже имеет метод **Clear**, очищающий текст, занесенный в окно редактирования, то вы можете в вашем переопределенном методе вызвать наследуемый метод **Clear** и добавить задание значения **FModified**. Чтобы точно знать, как выглядит объявление метода **Clear** в классе-предке, полезно обратиться к справке **C++Builder** и точно воспроизвести в вашем новом классе объявление переопределяемого метода. Это объявление в классе **TCustomEdit** имеет вид:

```
virtual void __fastcall Clear(void);
```

Таким образом, объявление и реализация переопределенного метода **Clear** может иметь вид:

```
class PACKAGE TEditLetNum : public TEdit
{
...
public:
...
    virtual void __fastcall Clear(void);
...
};

...
void __fastcall TEditLetNum::Clear(void)
{
    TEdit::Clear();           // Вызов родительского метода
    FModified = false;
}
```

Как видно из приведенного кода, для вызова родительского метода достаточно просто указать явным образом класс (в нашем случае **TEdit**), метод которого вызывается.

Впрочем, в данном случае вы могли бы и не обращаться к родительскому методу, а просто очистить текст в окне:

```
void __fastcall TEditLetNum::Clear(void)
{
    Text = «»;
    FModified = false;
}
```

Теперь запишем главную процедуру, ради которой мы и создавали свой компонент: процедуру, разрешающую или запрещающую ввод символов того или иного типа. Для этого нам надо анализировать вводимый пользователем символ. Это можно делать при обработке события **OnKeyPress**. Значит нам надо переопределить стандартную функцию генерации (т.е. стандартный обработчик) в родительском компоненте **TEdit**. Стандартные обработчики имеют то же имя, что и события, но без префикса **On**. То есть обработчик события **OnKeyPress** имеет имя **KeyPress**. Передаваемые в стандартный обработчик параметры те же, что вы можете видеть в заготовке процедуры обработки события, если щелкнете на этом событии в Инспекторе Объектов, но без параметра **Sender**. Например, щелкнув в Инспекторе Объектов на событии **OnKeyPress** любого компонента, вы увидите, что в процедуру обработки передаются параметры:

```
(TObject *Sender, char &Key)
```

Значит в создаваемом нами обработчике и при вызове родительского обработчика надо использовать только один параметр: **char &Key**.

Впрочем, поскольку вам надо будет точно воспроизвести объявление переопределяемой функции в классе-предке, обратитесь к справке C++Builder. Вы увидите, что функция **KeyPress** наследуется из класса **TWinControl** и объявлена в нем следующим образом:

```
DYNAMIC void __fastcall KeyPress(char &Key);
```

Это объявление надо в точности воспроизвести в своем классе-наследнике. Например, если вы пропустите в объявлении спецификатор **DYNAMIC**, вы получите при компиляции сообщение:

```
[C++ Error] EditLetNum.h(24): E2113 Virtual function '_fastcall TEditLetNum::KeyPress(char &)' conflicts with base class 'TWinControl'
```

Его смысл заключается в том, что возник конфликт с функцией **KeyPress**, объявленной в базовом классе **TWinControl**. В результате компиляция не будет завершена.

Таким образом, введение в модуль компонента требуемой нам функции может свестись к включению в раздел класса **protected** приведенного выше объявления и написанию в файле модуля реализации функции:

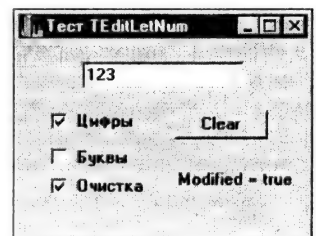
```
void __fastcall TEditLetNum::KeyPress(char &Key)
{
    Set <char, '0', '9'> Dig;
    Dig << '0' << '1' << '2' << '3' << '4' << '5'
        << '6' << '7' << '8' << '9';
    if ((! FEnableNum) && (Dig.Contains(Key)))
        Key = 0;
    if ((! FEnableLet) && !(Dig.Contains(Key)))
        Key = 0;
    if (Key != 0) FModified = true;
    TEdit::KeyPress(Key);
}
```

Эта функция сначала заменяет недопустимые символы нулевыми. Если символ допустимый, то задается значение **true** полю свойства **FModified**. В конце вызывается метод **KeyPress** родительского класса.

Можете скомпилировать (быстрая кнопка **Compile** — слева на рис. 7.2.) и протестировать полученный компонент. Проще всего это сделать, создав группу проектов (см. раздел 2.4.3), включающую ваш пакет и тестирующее приложение. Выполните команду **View | Project Manager** и перед вами откроется окно Менеджера Проектов, показанное на рис. 2.17 в главе 2. В нем будет вершина, соответствующая вашему пакету. С помощью кнопки **New** вы можете включить в группу новый проект, выбрав в окне **New Items** пиктограмму **Application**. В этом проекте и создавайте тестирующее приложение, возможный вид которого представлен на рис. 7.5. Форма содержит компонент **EditLetNum**, кнопку **Button** с надписью **Clear** (при ее нажатии выполняется метод **Clear**), метку **Label**, в которой отображается значение свойства **Modified** компонента **EditLetNum1**, и три индикатора типа **TCheckBox**.

Рис. 7.5

Тестовое приложение вашего нового компонента



Два из них (назовите их **CBNum** и **CBLet**) указывают допустимость ввода цифр и букв. О третьем речь пойдет в следующем разделе.

Обработчик события **OnCreate** формы может иметь вид:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    if (EditLetNum1->Modified)
        Labell->Caption = «Modified = true»;
    else Labell->Caption = «Modified = false»;
}
```

Этот обработчик заносит в метку **Labell** сообщение о текущем значении свойства **Modified**.

В обработчике события **OnKeyUp** компонента **EditLetNum** напишите оператор:

```
FormCreate(Sender);
```

Этот оператор вызывает тот же приведенный выше обработчик события **OnCreate** формы для отображения текущего значения **Modified**.

Обработчик события **OnClick** индикатора **CBNum** может иметь вид:

```
EditLetNum1->EnableNum = CBNum->Checked;
CBLet->Checked = EditLetNum1->EnableLet;
EditLetNum1->SetFocus();
```

Первый оператор этого обработчика устанавливает значение параметра **EnableNum** в зависимости от состояния индикатора **CBNum**. Второй оператор устанавливает состояние индикатора **CBLet** равным значению параметра **EnableLet**. Это надо, поскольку если, например, сбрасывается в **false** значение параметра **EnableNum**, а значение параметра **EnableLet** в этот момент тоже было равно **false**, то компонент **EditLetNum1** установит значение **EnableLet** равным **true**. И надо, чтобы это новое значение отобразилось в индикаторе **CBLet**.

Аналогично может выглядеть обработчик события **OnClick** индикатора **CBLet**:

```
EditLetNum1->EnableLet = CBLet->Checked;
CBNum->Checked = EditLetNum1->EnableNum;
EditLetNum1->SetFocus();
```

В обработчике события **OnClick** кнопки **Clear** напишите:

```
EditLetNum1->Clear();
FormCreate(Sender);
```

Эти операторы проверяют работу введенного вами метода **Clear** и отобразят на экране изменение значения свойства **Modified**.

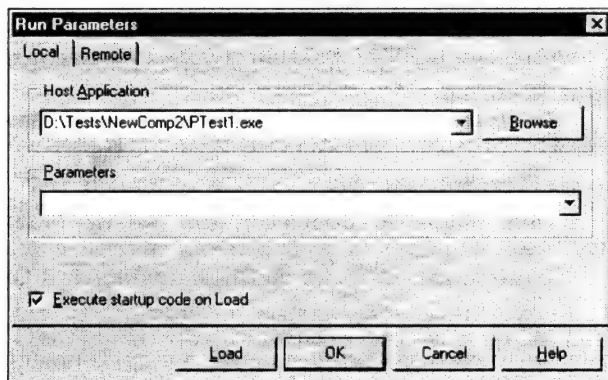
Оттранслируйте компонент. В вашем тестовом приложении установите свойства **EnableNum** и **EnableLet** в соответствии с тем, какие исходные установки индикаторов **CBNum** и **CBLet** вы задали. Активизируйте в окне Менеджера Проектов вершину вашего тестового приложения (см. раздел 2.4.3), запустите его на выполнение и проверьте в работе при различных значениях свойств **EnableNum** и **EnableLet**.

Если в написанных файлах вашего нового компонента что-то не в порядке, то вам может захотеться ввести в файл реализации компонента какие-то точки прерывания. Но это у вас не получится. При запуске теста на выполнение точки прерывания, введенные в файл компонента, окажутся недоступными. А если вы активизируете в окне Менеджера Проектов вершину пакета и попытаете выполнить его, то получите сообщение: «Cannot debug project unless a host application is defined. Use Run|Parameters... dialog box». Это означает, что вы сначала с помощью команды **Run | Parameters** должны определить хост — тестирующее приложение. Выполните эту команду, предварительно активизировав в окне Менеджера Проектов вершину пакета. Перед вами откроется диалог, представленный на рис. 7.6. В верхнем окошке

вы должны указать файл тестирующего приложения. Для этого можете воспользоваться кнопкой поиска Browse. Кнопка Load позволяет вам загрузить тестирующее приложение и начать его выполнение по шагам. Затем можете запускать выполнение обычной командой Run | Run (F9). Впрочем, после того, как вы один раз задали хост для пакета в окне рис. 7.6, вы можете в дальнейшем запускать пакет на выполнение обычным образом.

Рис. 7.6

Задание приложения, тестирующего DLL



7.3.5 Создание событий

В нашем примере требуется ввести событие **OnClear**, происходящее в момент очистки окна методом **Clear**. В C++Builder событие — это просто специальное свойство, являющееся указателем функции. Тип обобщенного указателя на функцию, которой передается один параметр типа **TComponent** (обычно **this**), — **TNotifyEvent**. Подобный тип используется в C++Builder для событий типа **OnClick** и многих других, которые передают в обработчик только один параметр — **TObject *Sender**. Подойдет этот тип и нам для события **OnClear**.

В этом случае объявления нашего события могут иметь вид:

```
private:
...
TNotifyEvent FOnClear;
...
__published:
...
__property TNotifyEvent OnClear = {read=FOnClear,
                                   write=FOnClear};
```

Эти объявления создают поле **FOnClear** типа **TNotifyEvent**, соответствующее событию. А само событие объявляется точно так же, как любое свойство. Только в **read** и **write** записываются не функции чтения и записи, а само поле.

Остается только вызвать в нужный момент обработчик события пользователя, если пользователь его предусмотрел. Проверка, имеется ли обработчик пользователя, осуществляется проверкой соответствующего события как булевой величины, возвращающей **true**, если пользователь предусмотрел свой обработчик. В нашем случае эта проверка и вызов обработчика пользователя осуществляется добавлением в начало написанной ранее процедуры метода **Clear** оператора:

```
if (OnClear) OnClear(this);
```

который вызывает обработчик пользователя **OnClear**.

Внесите указанные изменения в файлы компонента, откомпилируйте их, выделите на форме тестового приложения компонент **EditLetNum1** и посмотрите страницу событий в Инспекторе Объектов. Вы увидите, что в списке его событий

появилось событие **OnClear**. Дважды щелкните на его правом окне и вы увидите в тексте модуля заготовку для обработчика этого события:

```
void __fastcall TForm1::EditLetNum1Clear(TObject *Sender)
{
}
```

Внесите в этот обработчик какой-нибудь оператор, который бы отображал на экране факт свершения этого события, например:

```
ShowMessage("Событие OnClear");
```

Проверьте ваше тестовое приложение в работе.

Мы рассмотрели простейший вариант задания события типа **TNotifyEvent**. Попробуем несколько усложнить это событие. Пусть мы будем передавать в это событие параметр **CanClear**, который по умолчанию равен **true**, что будет означать разрешение очистить текст окна редактирования. Но предоставим пользователю возможность в своем обработчике события задать этому параметру значение **false**, что будет означать отказ от очистки.

Чтобы решить эту задачу, вам надо объявить новый тип события. Это объявление делается с помощью ключевого слова **__closure**. Пусть, например, вы хотите дать вводимому новому типу событий имя **TClear**. Тогда объявление указателя на этот тип может иметь вид:

```
typedef void __fastcall (__closure *TClear)
    (System::TObject *Sender, bool& CanClear);
```

Приведенный код объявляет указатель на функцию, принимающую два параметра: традиционный для всех обработчиков параметр **Sender** и булев параметр **CanClear**, который пользователь сможет изменять.

Приведенное выше объявление типа должно помещаться перед описанием класса. Тогда в объявлениях свойства и его поля надо ссылаться на этот тип, следующим образом изменив соответствующие операторы:

```
TClear FOnClear;
...
__property TClear OnClear = {read=FOnClear, write=FOnClear};
```

Осталось изменить метод **Clear** так, чтобы в нем учитывался новый параметр **CanClear**:

```
void __fastcall TEditLetNum::Clear()
{
    bool CanClear = true;
    if (OnClear) OnClear(this, CanClear);
    if (CanClear)
    {
        TEdit::Clear(); // Вызов родительского метода
        FModified = false;
    }
}
```

В этой функции вводится булева переменная **CanClear**. При наличии обработчика пользователя он вызывается и **CanClose** передается в этот обработчик в качестве второго параметра. Затем в зависимости от значения **CanClose** вызывается или не вызывается родительский метод очистки **OnClear**.

Теперь ваш компонент завершен. Полный текст его файлов имеет следующий вид.

Файл EditLetNum.h:

```
...
// Объявление типа события
typedef void __fastcall (__closure *TClear)
    (System::TObject *Sender, bool& CanClear);
```

```

class PACKAGE TEditLetNum : public TEdit
{
private:
// Закрытые элементы-данные класса
    bool FEnableLet;
    bool FEnableNum;
    bool FModified;
    TClear FOnClear;
protected:
// Защищенные методы записи
    void __fastcall SetEnableLet(bool AEnableLet);
    void __fastcall SetEnableNum(bool AEnableNum);
// Переопределение метода KeyPress
    DYNAMIC void __fastcall KeyPress(char &Key);
public:
// Объявление конструктора
    __fastcall TEditLetNum(TComponent* Owner);
// Объявление метода Clear
    virtual void __fastcall Clear(void);
// Свойство только времени выполнения
    __property bool Modified = {read=FModified, default=false};

    __published:
// Свойства компонента, включаемые в Инспектор Объектов
    __property bool EnableLet = {read=FEnableLet,
                                write=SetEnableLet,
                                default=true};
    __property bool EnableNum = {read=FEnableNum,
                                write=SetEnableNum,
                                default=true};
    __property TClear OnClear = {read=FOnClear, write=FOnClear};
};
...

```

Файл EditLetNum.cpp:

```

...
static inline void ValidCtrCheck(TEditLetNum *)
{
    new TEditLetNum(NULL);
}
//-----
__fastcall TEditLetNum::TEditLetNum(TComponent* Owner)
    : TEdit(Owner)
{
    FEnableLet = true;
    FEnableNum = true;
    FModified = false;
}
//-----
namespace Editletnum
{
    void __fastcall PACKAGE Register()
    {
        TComponentClass classes[1] = {__classid(TEditLetNum)};
        RegisterComponents(«MyComponents», classes, 0);
    }
}
//-----
void __fastcall TEditLetNum::SetEnableNum(bool AEnableNum)
{
    //Присваивание значения полю FEnableNum
    FEnableNum = AEnableNum;
}

```

```

// Если значения FEnableNum и FEnableLet = false,
// то полю FEnableLet присваивается true
if (! AEnableNum)
    if (! FEnableLet) FEnableLet = true;
}
//-----
void __fastcall TEditLetNum::SetEnableLet(bool AEnableLet)
{
    //Присваивание значения полю FEnableLet
    FEnableLet = AEnableLet;

    // Если значения FEnableNum и FEnableLet = false,
    // то полю FEnableNum присваивается true
    if (! AEnableLet)
        if (! FEnableNum) FEnableNum = true;
}
//-----
void __fastcall TEditLetNum::Clear(void)
{
    bool CanClear = true;
    if (OnClear) OnClear(this, CanClear);
    if (CanClear)
    {
        TEdit::Clear(); // Вызов родительского метода
        FModified = false;
    }
}
//-----
void __fastcall TEditLetNum::KeyPress(char &Key)
{
    Set <char, '0', '9'> Dig;
    Dig << '0' << '1' << '2' << '3' << '4' << '5'
        << '6' << '7' << '8' << '9';
    if ((! FEnableNum) && (Dig.Contains(Key)))
        Key = 0;
    if ((! FEnableLet) && !(Dig.Contains(Key)))
        Key = 0;
    if (Key != 0) FModified = true;
    TEdit::KeyPress(Key);
}

```

Откомпилируйте этот код. Теперь надо изменить тестирующее приложение. Возможный вид его уже был представлен ранее на рис. 7.5. Только раньше не пояснялось назначение еще одного индикатора **CheckBox** (назовем его **CBClear**), около которого написано «Очистка». Он предназначен для задания значения **CanClear** в обработчике события **OnClear**. Поэтому по сравнению с прежним приложением надо изменить обработчик **OnClear**, добавив в него анализ того, установлен флажок в индикаторе **CBClear**, или нет (анализ свойства **Checked**). Но тут вы столкнетесь с трудностью, если тестовое приложение делаете не заново, а изменяя предыдущее. По сравнению с прошлым вариантом компонента в функцию обработчика добавился параметр. Поэтому прежний обработчик события **OnClear** будет выдавать вам синтаксические ошибки. Надо или уничтожить его и написать новый, или вручную исправить заголовки в объявлении обработчика и в его реализации, которая может иметь вид:

```

void __fastcall TForm1::EditLetNum1Clear(TObject *Sender,
                                         bool &CanClear)
{
    ShowMessage("Событие OnClear");
    CanClear = CBClear->Checked;
    EditLetNum1->SetFocus();
}

```

В этом обработчике значение параметра **CanClear**, разрешающего или запрещающего очистку, задается в зависимости от того, установлен или нет индикатор **CBClear**.

Вот и все. Не забудьте только согласовать исходные значения параметров **EnableNum** и **EnableLet** с исходными состояниями индикаторов **CBNum** и **CBLet**. Если, например, вы установили **EnableNum** в **true**, а **EnableLet** в **false**, то индикатор **CBNum** должен быть включен (свойство **Checked = true**), а индикатор **CBLet** — выключен.

Запустите приложение и убедитесь в правильной работе вашего компонента.

7.3.6 Автоматизация разработки классов в C++Builder 5

Выше было рассмотрено, как вводить новые свойства и методы в компонент вручную. Однако в C++Builder 5 появился инструмент, который помогает автоматизировать этот процесс. Этот инструмент встроен в окно Исследователя Классов **ClassExplorer** (см. раздел 2.5.3.2). Опробуйте его в работе. Чтобы не портить созданный вами полезный компонент, начните все с начала: создайте новый пакет и в нем — новый компонент (см. раздел 7.3.1). А теперь попробуйте создать в нем новые свойства, методы события, пользуясь возможностями Исследователя Классов.

Если окно Исследователя Классов у вас не открыто, выполните команду **View | ClassExplorer**. Щелкните в окне **ClassExplorer** правой кнопкой мыши. Во всплывшем меню вы увидите новые разделы контекстного меню: **New Field** (новое поле), **New Property** (новое свойство), **New Method** (новый метод). Давайте начнем с раздела **New Property** и попробуем с его помощью ввести в класс компонента свойство **EnableLet**, которое мы вводили раньше вручную. После выбора раздела **New Property** перед вами откроется окно, представленное на рис. 7.7. Введите в нем в верхнее окошко **Property Name** имя свойства — **EnableLet**. В выпадающем списке **Type** выберите тип вводимого свойства — в нашем случае тип **bool**. В выпадающем списке **Add to Class** вы можете выбрать класс, объявленный в проекте, в который вводится свойство. Группа радиокнопок **Visibility** задает доступ к свойству: **public**, **private**, **protected** или **published**. В нашем случае надо включить кнопку **published**.

Рис. 7.7

Окно задания нового свойства

Установка индикатора `create field` приводит к автоматическому созданию в классе зарытого (**private**) поля, соответствующего вводимому свойству. Имя поля по умолчанию соответствует имени свойства с добавленным впереди символом 'F'. Впрочем, вы можете изменять это имя, как вам угодно.

Чтение свойства может быть задано несколькими способами. Если вы установите индикатор `use this field for the read specifier` (установив предварительно `create field`), то значение свойства будет определяться значением созданного поля без введения в класс функции чтения. Вы можете вместо этого установить индикатор `create Get method`. Тогда в класс будет включено объявление закрытого (**private**) метода чтения. При этом надпись индикатора `use this field for the read specifier` сменится на `use this field for implementing the Get method` — использовать данное поле для реализации функции чтения. Если вы установите этот индикатор, то в созданный метод чтения автоматически занесется оператор, возвращающий значение поля. Третья возможность задания функции чтения, если она вами уже реализована или если результат должен быть равен значению какого-то из ранее введенных полей (переменных) — выбрать функцию или поле из выпадающего списка `Reads`.

Одна из указанных выше возможностей должна быть реализована вами. Иначе кнопка ОК окна рис. 7.7 не станет доступна и вы не сможете завершить работу по созданию свойства.

В нашем случае мы не собираемся создавать специальную функцию чтения. Поэтому надо включить индикатор `use this field for the read specifier`.

Запись значения свойства также может быть задана несколькими способами. Если вы, установив предварительно `create field`, установите затем индикатор `use this field for the write specifier` (на рис. 7.7 этот идентификатор не виден, так как реализован другой вариант записи), то значение свойства будет равно значению созданного поля без введения в класс функции записи. Если вы вместо этого установите индикатор `create Set method`, то в класс будет включено объявление закрытого (**private**) метода записи. При этом, как и для случая чтения, надпись индикатора `use this field for the write specifier` сменится на `use this field for implementing the Set method` — использовать данное поле для реализации функции записи. Если вы установите этот индикатор, то в созданный метод записи автоматически занесется вариант реализующего его оператор, который будет рассмотрен позднее. Третья возможность задания функции записи, если она вами уже реализована или если результат должен быть равен значению какого-то из ранее введенных полей (переменных) — выбрать функцию или поле из выпадающего списка `Writes`. Если вы не воспользуетесь ни одним из этих способов задания записи значения свойства, то свойство будет только для чтения.

Если при задании способов чтения и записи свойства вы используете списки `Reads` и `Writes`, вы можете предварительно кнопкой с многоточием около надписи `Implement` выбрать файл, в котором реализованы функции, если это не тот файл, в котором объявлен класс. Можно также просто исправить имя файла, щелкнув курсором на его имени около этой надписи.

Если при задании функций чтения или записи вы воспользовались индикаторами `create Get method` или `create Set method`, вам станет доступен соответственно индикатор `Implement Get using member` или `Implement Set using member`. Включив его, вы можете выбрать реализацию из выпадающего списка данных-элементов и функций-элементов класса.

В нашем случае следует включить индикатор `create Set method` и индикатор `use this field for implementing the Set method`.

Окошко `Array` используется для задания свойства типа массива. Окошко `Index` используется для индексированных свойств. В окошко `Stored` может заноситься значение директивы спецификации класса памяти. Окошко `Default` используется для задания в свойстве директивы запоминания **default**.

После того, как вы установили в окне рис. 7.7 всю необходимую информацию, можете щелкнуть на ОК для выхода из окна или на Apply для внесения в код заданных установок и продолжения работы в диалоговом окне. В окне Исследователя Классов отобразятся созданные вами поле, свойство и функция записи.

В нашем случае это будет выглядеть так. В заголовочный файл в объявление класса внесутся операторы:

```
class PACKAGE TEditLetNum : public TEdit
{
private:
    bool FEnableLet;
    void __fastcall SetEnableLet(bool value);
    ...
__published:
    __property bool EnableLet = { read=FEnableLet, write=SetEnableLet,
                                default=true };
    ...
};
```

Как видите, объявления, которые мы раньше записывали вручную, создались автоматически. Отличие только в том, что функция записи включилась в раздел **private**, а мы вносили ее в раздел **protected**. Если требуется, ее объявление легко перенести в **protected** вручную.

В файл реализации занесутся операторы:

```
void __fastcall TEditLetNum::SetEnableLet(bool value)
{
    if(FEnableLet != value) {
        FEnableLet = value;
    }
}
```

Это заготовка реализации функции записи с уже внесенным в нее оператором записи. Конечно, вы можете дополнить этот оператор необходимыми действиями или вообще вместо автоматически созданного оператора ввести свои операторы. Появление оператора в реализации функции связано с тем, что в окне 7.7 был включен индикатор use this field for implementing the Set method. Если бы мы его не включали, то заготовка реализации функции записи имела бы вид:

```
void __fastcall TEditLetNum::SetEnableLet(bool value)
{
    //TODO: Add your source code here
}
```

В этом случае на месте, где должен быть код реализации функции, занесся бы оператор **To-Do List** (см. раздел 2.4.4.1). Он обеспечит отображение соответствующей строки в окне To-Do List, которая будет напоминать вам о необходимости завершить кодирование функции **SetEnableLet**.

Теперь рассмотрим другой раздел всплывающего меню ClassExplorer — **New Method** (новый метод). Используем его для объявления в нашем компоненте метода **Clear**. При выборе раздела New Method перед вами откроется окно, показанное на рис. 7.8.

В верхнем окошке Method Name вы должны написать имя создаваемого метода — в нашем случае **Clear**. В выпадающем списке Add to Class вы можете выбрать класс, объявленный в проекте, в который вводится свойство. В окошко Arguments вам надо занести список аргументов функции в том виде, в котором он должен появиться в скобках ее объявления. Радиокнопки группы Method Type позволяют задать тип создаваемого метода: Function — функция, Constructor — конструктор, Destructor — деструктор.

Рис. 7.8

Окно задания нового метода

Add Method

Method Name:

Add to Class:

Arguments:

Method type:

☒ Function ☐ Constructor ☐ Destructor

Function Result:

Visibility:

☒ Public ☐ Private ☐ Protected ☐ Published

Directives:

☐ abstract (1) ☐ const (3)

☒ virtual (2) ☒ _fastcall (4)

☐ Message Handler:

Implementation details:

☒ Call inherited ☐ Inline ☐ Implicit inline

File: D:\Tests\NewComp2\TEditLetNum2.cpp

Reset (R) OK Cancel Apply

Если вы включили кнопку **Function**, то становится доступным выпадающий список **Function Result**, позволяющий задать тип возвращаемого функцией значения (в нашем случае **void**). Для конструкторов и деструкторов этот список недоступен, поскольку эти методы не возвращают никаких значений.

Если вы включили кнопку **Constructor** или **Destructor**, то имя в окошке **Method Name** автоматически заменится на принятое соответственно для конструктора или деструктора.

Группа радиокнопок **Visibility** задает доступ к методу: **public**, **private**, **protected** или **published**.

Группа индикаторов **Directives** позволяет задать спецификаторы объявления метода. Их смысл ясен из надписей около индикаторов. Если вы включите индикатор **Message Handler**, это будет означать, что вы хотите создать обработчик сообщения **Windows**. Тогда станет доступен выпадающий список, содержащий перечень сообщений **Windows**, в котором вы можете выбрать нужное сообщение. Пример, использующий индикатор **Message Handle**, приведен в разделе 6.3.3.

Группа индикаторов **Implementation details** позволяет установить особенности реализации метода. Индикатор **Call inherited** позволит автоматически внести в реализацию метода оператор вызова наследуемого метода с тем же именем. Индикатор **Inline** обеспечивает создание функции со спецификацией **inline** — встраиваемая. При включении этого индикатора становится доступным индикатор **Implicit Inline**. Если этот индикатор оставить не включенным, то в объявление функции в классе добавится спецификатор **inline**, а реализация функции буде помещена в файл реализации модуля. Если же вы включите индикатор **Implicit Inline**, то описание функции будет совмещено с ее объявлением в классе. Вообще говоря, это плохой стиль программирования, так как следует избегать смешения открытого интерфейса класса, содержащегося в его объявлении, и реализации класса.

Давайте посмотрим, к чему приводит работа с описанным окном. При установках, показанных на рис. 5.8, в описание класса компонента будет включено объявление нового открытого метода:

```
class PACKAGE TEditLetNum : public TEdit
{
...

```



```
public:
    ...
    virtual void __fastcall Clear();
    ...
};
```

В файл реализации будет включена заготовка нового метода:

```
void __fastcall TEditLetNum2::Clear()
{
    TEdit::Clear();
}
```

Как видите, в реализацию уже внесен оператор вызова метода базового класса. Это следствие включенного в окне рис. 7.8 индикатора `Call inherited`. Если бы этот индикатор не был включен, то вместо этого оператора появилась бы строчка:

```
//TODO: Add your source code here
```

Это оператор **To-Do List** (см. раздел 2.4.4.1). Он обеспечивает отображение соответствующей строки в окне To-Do List, которая будет напоминать вам о необходимости завершить кодирование функции.

Рассмотрим некоторые дополнительные возможности окна рис. 7.8. Если бы вы задали создание некоторой защищенной функции **FProtect** с возвращаемым типом `void` и включили индикатор `inline`, не включая `Implicit Inline`, то в раздел **protected** класса включилось бы объявление

```
void inline FProtect();
```

а в файл реализации модуля записался бы текст:

```
void inline TForm1::FProtect()
{
    //TODO: Add your source code here
}
```

т.е. была бы создана встраиваемая функция, в которой ее объявление отделено от описания. А если бы для этой функции вы включили индикатор `Implicit Inline`, то для нее в раздел **protected** объявления класса включился бы код:

```
protected:
    void FProtect()
    {
        //TODO: Add your source code here
    }
```

Таким образом, в этом случае функция также была бы объявлена как встраиваемая, но ее объявление и реализация были бы объединены.

Мы рассмотрели команды `New Property` и `New Method` контекстного меню `ClassExplorer`. Команда `New Field` позволяет ввести в описание класса новое поле. Она очень проста, так что ее использование не вызовет у вас проблем.

Рассмотренные команды `ClassExplorer` показывают, какой прекрасный инструмент подарила нам фирма Borland в `C++Builder 5`, введя новые возможности в Исследователь Классов. Он, несомненно, позволит экономить нам немало времени при разработке новых классов и избегать множества случайных ошибок.

7.4 Депозитарий — хранилище форм, фреймов и проектов

В Депозитарий (хранилище — `Repository`) вы попадаете, когда выполняете команду `File | New`. При этом открывается диалоговое окно `New Items`, в котором вы можете выбрать включенные в `C++Builder` готовые формы или воспользоваться разработанными фирмой Borland мастерами. Все эти возможности подробно рассмот-

рены в главе 2. А в данном разделе мы остановимся на использовании Депозитария для хранения собственных разработок.

Нередко создание сложной формы с множеством размещенных на ней компонентов требует немалого времени. Причем однажды разработанная удачная форма может пригодиться вам в последующих приложениях. Конечно, можно сохранить ее в каком-либо каталоге и, когда возникнет необходимость, использовать в очередном проекте. Но если разработка этого нового проекта будет не скоро, вы, возможно, потратите много времени на поиск каталога с необходимой вам формой, если вообще найдете ее. Хотелось бы иметь возможность как-то зарегистрировать свои удачные разработки в C++Builder, чтобы в дальнейшем без труда повторно их использовать. Такую возможность и предоставляет вам Депозитарий. Кроме того в Депозитарии можно хранить фреймы (см. раздел 3.7.8) — фрагменты, включающие в себя какие-то компоненты.

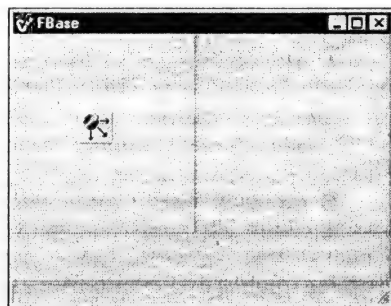
Депозитарий позволяет не просто хранить формы, но и наследовать их, т.е. создавать иерархию форм. Это важно, поскольку в сложном приложении, содержащем много форм, все эти формы должны быть спроектированы в едином стиле, с единообразным расположением органов управления, ввода и редактирования данных, в единой цветовой гамме и т.п. Это легко делается созданием иерархии форм.

Пусть, например, вы решили, что основные окна вашего приложения должны иметь вид, представленный на рис. 7.9. Окна состоят из двух панелей с перемещаемыми границами, в которых будут размещаться окна редактирования и списки, ниже них — панель, в которой будут располагаться кнопки управления, а внизу окна должна быть панель состояния. На форме должен также размещаться невидимый компонент **ApplicationEvents**, в обработчике события **OnHint** которого записан оператор, обеспечивающий отображение подсказок в панели состояния (подробнее об этом см. в разделе 4.1.9). Все необходимые свойства панелей установлены, написаны операторы, обеспечивающие вывод на панель состояния подсказок. Задано имя формы (**Name**) — **FBase**.

Имеет смысл сохранить эту форму в Депозитарии и в дальнейшем использовать для создания на ее основе различных конкретных форм вашего приложения. Но перед этим сохраните модуль вашей формы. Для этого можно выполнить команду **File | Save As**. Сам проект можно не сохранять. При сохранении укажите имя файла **UFBase**.

Рис. 7.9

Вид основного окна приложения, включаемого в Депозитарий



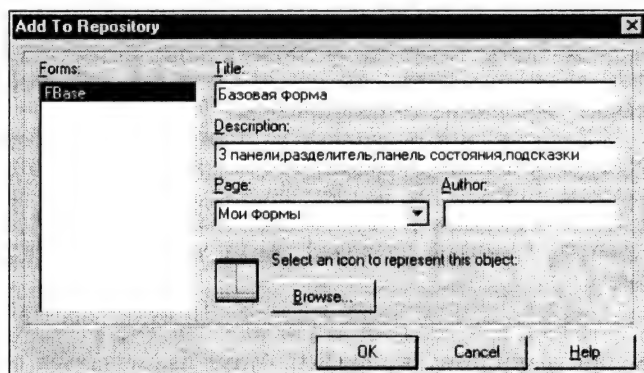
Записать форму в Депозитарий можно следующим образом. Прежде всего щелкните на вашей форме правой кнопкой мыши и выберите во всплывшем контекстном меню раздел **Add To Repository**. Откроется диалоговое окно, вид которого приведен на рис. 7.10. В верхнем окне **Title** вы должны написать название вашей формы — подпись под ее пиктограммой при входе в Депозитарий. В следующем окне — **Description** можете написать более развернутое пояснение. Его может увидеть пользователь, войдя в Депозитарий, щелкнув правой кнопкой мыши и выбрав

во всплывшем меню форму отображения View Details. В выпадающем списке Page вы можете выбрать страницу Депозитария, на которой хотите разместить пиктограмму своей формы. Впрочем, вы можете указать и новую страницу с новым заголовком (Мои формы на рис. 7.10). В результате она появится в Депозитарии.

В окне Author вы можете указать сведения о себе как об авторе. Наконец, если стандартная пиктограмма вас не устраивает, вы можете выбрать другую, щелкнув на кнопке Browse. После выполнения всех этих процедур щелкните на кнопке ОК и ваша форма окажется включенной в Депозитарий.

Рис. 7.10

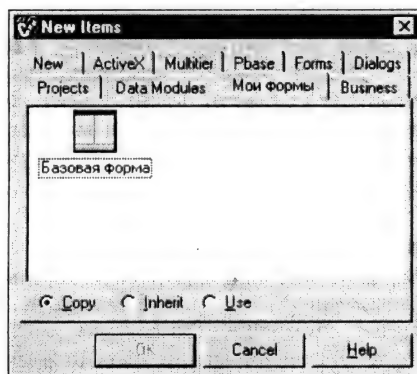
Окно добавления формы в Депозитарий



Теперь вы можете использовать ее в последующих ваших приложениях. Для этого вам надо будет выполнить команду File | New и в открывшемся диалоговом окне New Items отыскать вашу форму (рис. 7.11).

Рис. 7.11

Окно New Items с включенной новой формой



В нижней части окна расположены три радиокнопки, которые определяют, как именно вы хотите заимствовать форму из Депозитария: Copy — копировать, Inherit — наследовать, Use — использовать. Если включена кнопка Copy, то файлы формы просто будут скопированы в ваше приложение. При этом никакой дальнейшей связи между исходной формой и копией не будет. Вы можете спокойно изменять свойства вашей копии и это никак не отразится на форме, хранящейся в Депозитарии. А если вы в дальнейшем что-то измените в форме, хранящейся в Депозитарии, то эти изменения никак не затронут вашего приложения, куда вы до этого скопировали форму.

При включенной кнопке Inherit вы получите в своем проекте форму, наследующую размещенной в Депозитарии. Это значит, что если вы что-то измените в форме, хранящейся в Депозитарии, то это отразится при перекомпиляции во всех про-

ектах, которые наследуют эту форму. Однако, изменения в наследуемых формах никак не скажутся на свойствах формы, хранящейся в Депозитарии.

При включенной кнопке **Use** вы получите режим использования. В этом случае в ваш проект включится сама форма, хранящаяся в Депозитарии. Значит любое изменение свойств формы, сделанное в вашем проекте, отразится и на хранящейся в Депозитарии форме, и на всех проектах, наследующих или использующих эту форму (при их перекомпиляции).

Таким образом, режим **Inherit** целесообразно использовать для всех модулей вашего проекта, а режим **Use** — для изменения базовой формы. Тогда усовершенствование вами базовой формы будет синхронно сказываться на всех модулях проекта при их перекомпиляции.

Давайте построим приложение, которое позволит все это продемонстрировать. Если вы проделали процедуры по включению формы в Депозитарий, то можете приступать к построению тестового проекта. А если нет — проделайте эти процедуры сейчас. Если не хочется делать форму такую, как было описано, сделайте более простую, но содержащую 2 панели — они нам потребуются для дальнейшего. Только сотрите надписи на них (**Caption**).

Итак, построим проект, демонстрирующий различные режимы извлечения формы из Депозитария.

1. Откройте новый проект.
2. Выполните команду **Project | Remove from Project** и удалите модуль **Unit1** и его форму из проекта.
3. Выполните команду **File | New** и в окне **New Items** отыщите вашу форму. Включите индикатор **Copy** (он включен по умолчанию) и щелкните на кнопке **OK**.
4. Измените имя новой формы вашего приложения на **FCopy** (переименование формы необходимо, т.к. без этого далее невозможно будет заимствовать из Депозитария аналогичные формы).
5. Повторите команду **File | New**, в окне **New Items** опять отыщите вашу форму. Включите индикатор **Inherit** и щелкните на **OK**.
6. Измените имя новой формы вашего приложения на **FInherit**.
7. Повторите в третий раз команду **File | New** и поиск в окне **New Items** вашей формы. На этот раз включите индикатор **Use** и щелкните на **OK**.

Теперь взгляните на имена модулей, появившихся в вашем проекте. Для двух первых форм модули имеют традиционные имена — **Unit1** и **Unit2**. А вот имя третьего модуля совпадает с тем, под которым вы сохранили модуль формы перед записью в Депозитарий — **UFBase**. Это значит, что **C++Builder** в режиме **Use** включил в ваше приложение исходный модуль хранящейся в Депозитарии формы. И, значит, любые изменения в нем приведут к изменению хранящейся формы.

Давайте сохраним проект, чтобы задать первым двум модулям более осмысленные имена.

8. Сохраните проект, задав имя файла первого модуля — **UCopy**, а имя второго — **UInherit**. Имя файла проекта можете задать любым.
9. Разнесите формы на экране так, чтобы видеть хотя бы заголовки каждой (например, каскадом).
10. Выполните команду **Project | Options**. Установите все формы проекта автоматически создаваемыми.
11. Во всех формах установите значение свойства **Visible** в **true**, чтобы при запуске приложения все они были видны.

Приложение завершено. Сохраните его и запустите. Вы увидите на экране три одинаковых окна, различающиеся только заголовками. Закройте приложение

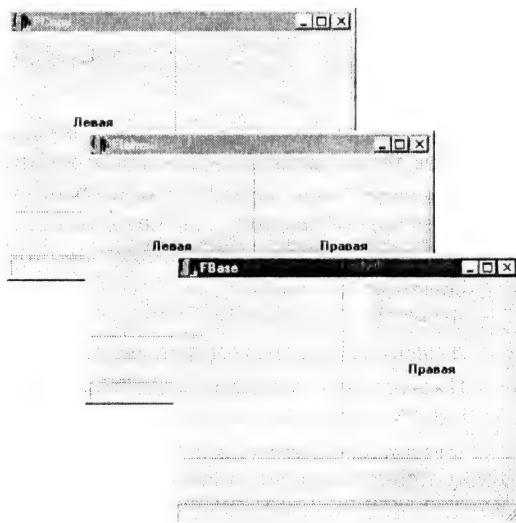
(окно **FCopy**). Измените что-нибудь в форме **FCopy**. Например, введите текст «Левая» в надпись (**Caption**) левой панели. Сохраните проект и запустите опять приложение. Вы увидите, что на форме **FCopy** появилась введенная вами надпись, а панели остальных форм по-прежнему пустые. Это очевидный результат, поскольку форма **FCopy** содержится в модуле **UCopy**, являющемся изолированным от всех остальных, хотя и создан как копия базового модуля **UBase**.

Закройте приложение, сотрите надпись на панели в форме **FCopy** и сделайте аналогичную надпись «Левая» на панели формы **FInherit**. Сохраните и запустите опять приложение. Вы увидите, что на форме **FInherit** появилась введенная вами надпись, а панели остальных форм по-прежнему пустые.

Опять закройте приложение и, не стирая надпись в левой панели формы **FInherit**, сделайте надпись «Правая» в правой панели третьей формы **FBase**. Сохраните и запустите приложение. Вы увидите (рис. 7.12), что на форме **FBase** появилась введенная вами надпись. А на форме **FInherit** имеются надписи на обеих панелях. Надпись на левой введена в модуле этой формы, а надпись на правой панели наследуется из базового модуля **UBase**. Таким образом, в наследуемых формах те свойства, которые были изменены в процессе проектирования, так и остаются неизменными. А остальные наследуются из базовой формы.

Рис. 7.12

Тестирование различных режимов
заимствования форм из Депозитария



В этом и заключаются положительные стороны наследования форм. Если вы построили различные формы вашего приложения (настоящего, а не этого тестового) наследованием, то изменив родительскую форму (например, изменив в ней размеры панелей или добавив какие-нибудь кнопки), вы автоматически внесете эти изменения и во все производные формы. Правда, эти изменения проявятся только при перекомпиляции вашего приложения.

Но у наследования есть и отрицательные свойства. Попробуйте в вашем приложении удалить в форме **FInherit** одну панель. Сделать это невозможно, так как C++Builder выдаст сообщение об ошибке. Однако, этот недостаток можно обойти. В производной форме можно сделать ненужные компоненты невидимыми (**Visible = false**) и недоступными (**Enabled = false**). Тогда они как бы исчезнут с формы.

Мы рассмотрели включение в Депозитарий форм и их использование. Точно так же, начиная с C++Builder 5, можно включать в Депозитарий и аналогичным образом использовать фреймы (см. главу 3 раздел 3.7.8). Это дает дополнительные

возможности организовывать иерархию не только форм, но и типовых фрагментов форм — панелей.

В Депозитарий можно включать не только формы и фреймы, но и целые проекты. Если вы хотите включить в него ваш проект, откройте его и выполните команду Project | Add To Repository. Дальнейшие действия аналогичны тем, которые вы выполняли при включении в Депозитарий формы. Отличие проекта от формы при их заимствовании из Депозитария состоит в том, что проект можно взять оттуда только в режиме Copy, т.е. скопировать его и далее сохранить под другим именем. Если вы хотите взять проект, хранящийся в Депозитарии, то начать работу надо не с привычной команды File | New Application, а с команды File | New. Вы выбираете проект из Депозитария и сразу же открывается диалоговое окно с предложением указать каталог, в котором вы хотите сохранить копию проекта. После этого можете обычным образом работать с этой копией и менять в ней все, что вам захочется.

Если какие-то формы и фреймы, хранящиеся в Депозитарии, стали вам не нужны, вы можете удалить их в диалоговом окне, открываемом при выполнении команды Tools | Repository (см. раздел 14.2.3 главы 14). В этом же окне вы можете при желании сделать свою форму или проект, хранящиеся в Депозитарии, соответственно формой или проектом по умолчанию.

7.5 Динамически присоединяемые библиотеки DLL

7.5.1 Назначение DLL

Динамически присоединяемая библиотека DLL — это еще одна возможность повторного использования разработанных вами кодов. DLL — это специального вида исполняемый файл с расширением .dll, используемый для хранения функций и ресурсов отдельно от исполняемого файла. Обычно, когда вы пишете программу и создаете функции, ресурсы и т. п., все они komponуются в ваш исполняемый файл. Это называется *статическим связыванием*. Когда же вы используете DLL, то вызываемые из нее функции используются вашим модулем в процессе его выполнения. DLL делает полезные, часто используемые функции доступными сразу для многих приложений одновременно, хотя работа ведется только с одной ее копией на диске и в памяти. Обычно DLL не загружается в память, пока это не станет необходимым, но, будучи однажды загружена, она делает свои функции и ресурсы доступными для любой программы.

В этой книге неявным образом уже многократно использовались DLL. Например, именно в DLL хранятся все процедуры и функции API Windows, которые очень часто применяются или непосредственно, или косвенно через аналогичные функции C++Builder, инкапсулирующие те или иные функции API Windows. В DLL хранятся также все стандартные диалоги Windows, которые вы, наверняка, постоянно используете чуть ли не в любом своем приложении. Так что с DLL вы уже знакомы очень тесно, хотя, может быть, и не подозреваете об этом. Но в этом разделе речь пойдет о других DLL — о библиотеках, создаваемых вами для хранения собственных кодов.

Предположим, вы разработали различные полезные процедуры и функции и используете их в различных своих приложениях. Конечно, вы можете просто скопировать эти процедуры и функции в каждый ваш проект. Но тогда, если несколько ваших приложений работает на компьютере одновременно, то копии ваших процедур загружены в память в выполняемом модуле каждого приложения и память, таким образом, расходуется неэффективно. Да и затраты пространства на дисках также избыточны, поскольку эти процедуры в составе выполняемых модулей также тиражируются. Если же вы поместили свои процедуры в DLL, то все эти проблемы снимаются.

Создание DLL повышает также гибкость вашей программы. Например, вы можете создать несколько библиотек, содержащих все используемые вашим приложением тексты — надписи, подсказки и т.п. Каждая из этих библиотек может содержать тексты на том или ином языке: русском, английском, немецком. Тогда в зависимости от того, какую из этих библиотек будет применять пользователь, программы будут общаться с ним на том или ином языке.

Еще одним преимуществом DLL является то, что они могут использоваться приложениями, написанными на других алгоритмических языках. Например, вы можете использовать библиотеки, написанные на Object Pascal, Visual Basic, Access Basic и др. А ваши библиотеки, созданные в C++Builder, смогут использовать любые системы, которые умеют вызывать DLL, независимо от того, на каких алгоритмических языках они написаны.

7.5.2 Статическое и динамическое присоединение DLL к приложению

Библиотеки DLL могут связываться с вашим приложением двумя путями: *статическим присоединением* или *динамическим присоединением*.

Статическое присоединение означает, что DLL загружается сразу, как только начинает выполняться приложение, которое будет ее использовать. Это наиболее простой способ использования DLL. Вызов функций в приложении, использующем подобную DLL, почти не отличается от вызова любых других функций. Некоторыми недостатками такого подхода можно считать увеличение времени загрузки вашего приложения (ведь кроме выполняемого модуля приложения в этот момент грузятся и модули соответствующих DLL) и невозможность выполнения приложения пользователем, у которого нет соответствующего файла DLL. Последнее трудно назвать существенным недостатком, поскольку, конечно, вы должны распространять приложение вместе с DLL. Но ведь некоторые действия приложение могло бы делать и без DLL, т.е. оно могло бы в урезанном виде функционировать и при отсутствии DLL. Однако, при статическом присоединении это невозможно. Еще одним недостатком статического присоединения является то, что DLL занимает память все время, в течение которого выполняется приложение, независимо от того, вызываются ли в данном сеансе работы с приложением какие-то функции библиотеки, или нет.

Статическое присоединение подразумевает, что для DLL создан специальный файл описаний импортируемых функций (import library file). Этот файл имеет то же имя, что и соответствующая DLL, и расширение **.lib**. Этот файл **.lib** должен быть связан с вашим приложением на этапе компиляции.

Динамическое присоединение отличается от статического тем, что библиотека DLL загружается только в тот момент, когда необходимо выполнить какую-то хранящуюся в ней функцию. Затем эту библиотеку можно выгрузить из памяти. Это обеспечивает, конечно, более эффективное использование памяти. Но зато вызов соответствующих функций библиотеки существенно усложняется, да и время вызова тоже увеличивается из-за необходимости загружать и выгружать библиотеку.

Для вызова библиотечной функции из библиотеки, присоединяемой подобным образом, надо прежде всего загрузить библиотеку функцией **LoadLibrary** API Windows. Затем с помощью функции **GetProcAddress** надо получить указатель на интересующую вас функцию библиотеки. Только после этого можно выполнять функцию. А затем с помощью функции **FreeLibrary** надо выгрузить библиотеку из памяти. Пусть, например, вы создали библиотеку **mydll.dll**, содержащую некоторую функцию **MyFunction()**. Тогда для использования этой функции при динамическом присоединении библиотеки вам надо предусмотреть в своем приложении следующее. Вы должны объявить указатель на эту функцию. Например:

```
void (__stdcall *MyFunction) (HWND)
```


Тогда операции с библиотекой могут строиться по следующей схеме:

```
// загрузка DLL
HINSTANCE dllInstance = LoadLibrary(«mudll.dll»);

// получение адреса функции
MyFunction = (void(__stdcall*)(HWND))
    GetProcAddress(dllInstance, «_MyFunction»);

// вызов функции
MyFunction(Application->Handle);

// выгрузка DLL
FreeLibrary(dllInstance);
```

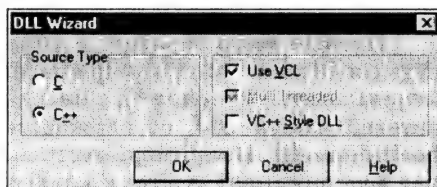
Как видите, все это достаточно громоздко. Так что если только нет острой необходимости использовать именно динамическое связывание, лучше всегда использовать статическое связывание.

7.5.3 Создание DLL

Создание своей DLL начинается с выполнения команды File | New и выбора в окне New Items на странице New пиктограммы DLL Wisard — Мастера DLL. Вы попадете в небольшое диалоговое окно, представленное на рис. 7.13. В нем вы можете выбрать язык DLL — C или C++. Индикатор Use VCL позволит вам создать DLL, которая может содержать компоненты библиотеки VCL. При этом в модуль включится файл **VCL.h** и установятся опции компоновки, обеспечивающие совместимость с объектами VCL. Индикатор Multi Threaded позволит работать с несколькими потоками (нитеями) выполнения. А индикатор VC++ Style DLL обеспечит создание DLL в стиле Microsoft Visual C++.

Рис. 7.13

Окно задания опций создания DLL



После установки всех опций и щелчка на ОК вы попадете в окно Редактора Кода, в котором появится (при установках, показанных на рис. 7.13) следующий текст:

```
#include <vcl.h>
#pragma hdrstop
//
// Important note about DLL memory management when your DLL
// uses the static version of the RunTime Library:
//
// If your DLL exports any functions that pass String objects
// (or structs/ classes containing nested Strings) as parameter
// or function results, you will need to add the library
// MEMMGR.LIB to both the DLL project and any other projects
// that use the DLL. You will also need to use MEMMGR.LIB
// if any other projects which use the DLL will be performing
// new or delete operations on any non-TObject-derived classes
// which are exported from the DLL. Adding MEMMGR.LIB to your
// project will change the DLL and its calling EXE's to use the
// BORLNDMM.DLL as their memory manager. In these cases, the
// file BORLNDMM.DLL should be deployed along with your DLL.
//
```

```
// To avoid using BORLNDMM.DLL, pass string information using
// «char *» or ShortString parameters.
//
// If your DLL uses the dynamic version of the RTL, you do not
// need to explicitly add MEMMGR.LIB as this will be done
// implicitly for you
```

(Комментарий гласит: Важное замечание об организации памяти DLL, если ваша DLL использует статическую версию библиотеки времени выполнения RunTime Library:

Если ваша DLL экспортирует какие-то процедуры или функции, которым передаются в виде параметров или от которых получаются как результат объекты строк String (или структуры и классы, содержащие вложенные строки), вы должны добавлять библиотеку MEMMGR.LIB как в проект DLL, так и в любой проект, использующий эту DLL. Вы должны также включать MEMMGR.LIB в любой проект, использующий операции создания (new) и удаления (delete) объектов экспортируемых из данной DLL классов, не наследующих TObject. Добавление MEMMGR.LIB в ваш проект изменит DLL и ее вызовы так, чтобы для распределения памяти использовалась библиотека BORLNDMM.DLL — диспетчер разделяемой памяти. В этих случаях файл BORLNDMM.DLL должен распространяться вместе с файлом вашей DLL. Чтобы избежать использования BORLNDMM.DLL, передавайте строковую информацию посредством параметров типа «char *» или ShortString. Если ваша DLL использует динамическую версию RTL, то вам не надо явным образом добавлять MEMMGR.LIB, так как это добавление будет сделано неявно.)

```
//-----
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
    return 1;
}
```

Прежде всего обратите в этом коде внимание на длинный комментарий. Его суть сводится к тому, что при передаче в функцию и из функции строк желательно использовать тип (char *), а не, например, **AnsiString**. Это избавит и вас, и пользователей вашей DLL от сложностей, связанных с необходимостью использовать **borlndmm.dll**. Например, пусть вы хотите включить в вашу DLL функцию, которая воспринимает строку и ключ типа **char**, и возвращает ее закодированной этим ключом. Кодировка проводится самая примитивная: сложением каждого символа строки с ключом по операции исключающее ИЛИ. Кстати, такая кодировка легко поддается декодированию: достаточно с закодированной строкой выполнить ту же операцию с тем же ключом, и вам вернется исходная строка.

Эту функцию можно реализовать следующим образом:

```
AnsiString Code(AnsiString s, char Key)
{
    for (int i = 1; i <= s.Length(); i++)
        s[i] = s[i] ^ Key;
    return s;
}
```

Но для включения в DLL лучше использовать другой тип строк:

```
char * Code(char *s, char Key)
{
    for (int i = 0; ; i++)
    {
        if (s[i] == '\0') break;
        s[i] = s[i] ^ Key;
    }
    return s;
}
```

Обе приведенные функции выполняют аналогичные операции, но вторая не вызывает никаких сложностей при включении ее в DLL.

Теперь обратите внимание в заготовке модуля DLL на строку

```
#include <vcl.h>
```

Она подключает заголовочный файл **vcl.h**. Этот файл нужен, если вы используете какие-то классы, формы, функции, связанные с библиотекой визуальных компонентов. В противном случае вы можете удалить этот оператор из текста DLL.

В конце приведенного текста заготовки DLL используется функция **DllEntryPoint**, необходимая для загрузки и выгрузки библиотеки. Она создает дескриптор **hinst** вашей DLL. Он требуется при выполнении некоторых функций, например, **LoadIcon**, **LoadCursor** и др. В этих случаях вы можете использовать параметр **hinst** для создания соответствующей глобальной переменной.

Для того, чтобы уяснить последовательность построения и использования DLL, давайте теперь создадим DLL с именем **MyDLL** и включим в нее приведенную выше функцию **Code**. Выполните последовательно следующие шаги.

1. Выполните команду **File | New** и выберите в окне **New Items** на странице **New** пиктограмму **DLL Wizard**. Сделайте в диалоговом окне установки, показанные на рис. 7.13. Вы попадете в окно Редактора Кода, куда будет загружен приведенный ранее текст.
2. Удалите из текста комментариев.
3. Вставьте перед описанием директиву включения головного файла:

```
#include «MyDLL.h»
```

Файла **MyDLL.h** пока нет, но скоро вы его создадите.

4. Вставьте после описания **DllEntryPoint** приведенное выше описание своей функции **Code**.
5. Сохраните ваш проект под именем **MyDLL**.

В результате рассмотренных действий файл **MyDLL.cpp** примет следующий вид:

```
#include <vcl.h>
#pragma hdrstop
//-----
#include «MyDLL.h»
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
    return 1;
}
//-----
char * Code_Dec(char *s, char Key)
{
    for (int i = 0; ; i++)
    {
        if (s[i] == '\0') break;
        s[i] = s[i] ^ Key;
    }
    return s;
}
```

Теперь давайте создадим заголовочный файл библиотеки.

6. Выполните команду **File | New** и выберите в окне **New Items** на странице **New** пиктограмму **Header File**. Вы попадете в окно Редактора Кода, куда будет загружен текстовый файл.
7. Запишите в файл следующий код:

```

#ifndef _MYDLL_H
#define _MYDLL_H

#ifdef __DLL__
# define DLL_EI __declspec(dllexport)
#else
# define DLL_EI __declspec(dllimport)
#endif

extern «C» char * DLL_EI Code_Dec(char *s, char Key);

#endif

```

8. Сохраните файл командой File | Save с именем MyDLL.h.

Давайте рассмотрим в приведенный текст заголовочного файла. Посмотрите на фрагмент, начинающийся с директивы **#ifdef**. Использованный в нем идентификатор **DLL_EI** может быть любым — это просто произвольный идентификатор. А логика работы данного фрагмента следующая: если определен идентификатор **__DLL__**, то идентификатор **DLL_EI** раскрывается как **__declspec(dllexport)**; если же **__DLL__** не определен, то идентификатор **DLL_EI** раскрывается как **__declspec(dllimport)**. C++Builder автоматически определяет **__DLL__** в случае, если создается проект DLL, и не определяет этот идентификатор при создании объекта приложения. Таким образом, в зависимости от того, включается ли заголовочный файл в библиотеку или в приложение, он будет выглядеть по-разному. При компиляции библиотеки строка, определяющая нашу функцию **Code_Dec**, после раскрытия макроса будет восприниматься как

```
extern «C» char * __declspec(dllexport) Code_Dec(char *s, char Key);
```

Здесь конструкция **__declspec(dllexport)** означает, что функция может экспортироваться из библиотеки, то есть может вызываться внешними приложениями. Подобным образом должны быть перечислены все функции DLL, предназначенные для прямого использования в приложениях. Помимо таких функций в DLL могут быть вспомогательные функции-утилиты, предназначенные только для использования другими функциями. Подобные утилиты не должны определяться как экспортируемые.

Когда тот же заголовочный файл включается в приложение, то эта же строка после раскрытия макроса имеет другой вид:

```
extern «C» char * __declspec(dllimport) Code_Dec(char *s, char Key);
```

Здесь конструкция **__declspec(dllimport)** означает, что функция импортируется, то есть вносится в модуль из DLL. Таким образом один и тот же заголовочный файл может использоваться и при создании DLL, и в приложениях, обращающихся к данной DLL.

Продолжим создание нашей DLL.

9. Выполните команду Project | Options и в окне опций проекта на странице Linker убедитесь, что включен индикатор опции **Generate import library**. Эта опция обеспечит при создании DLL автоматическую генерацию файл **.lib**, необходимого для описанного ранее статического присоединения DLL к проектам.

10. Выполните команду Project | **Build MyDLL**. В результате будут созданы файлы **MyDLL.dll** и **MyDLL.lib**.

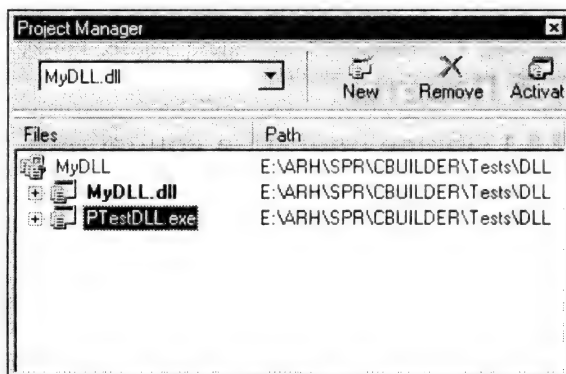
Если вы захотите теперь протестировать вашу DLL и выполните команду Run | Run (F9), то получите сообщение: «Cannot debug project unless a host application is defined. Use Run|Parameters... dialog box». Это означает, что вы сначала с помощью команды Run | Parameters должны определить хост — тестирующее приложение. Эта процедура уже была нами рассмотрена при тестировании нового компонента (см. раздел 7.3.4 рис. 7.6).

Тестирующее приложение вы можете строить обычным способом, независимо от вашей DLL. Но мы рассмотрим ниже более удобный путь — объединение с помощью Менеджера Проектов модуля DLL и тестирующего проекта в одну группу. Это позволяет более гибко переключаться между отлаживаемой библиотекой и кодом теста. Итак, продолжим нашу работу с DLL и напомним тестирующее приложение.

11. Выполните команду View | Project Manager. Перед вами откроется окно Менеджера Проектов, представленное на рис. 7.14, но пока только с одной вершиной MyDLL.
12. Нажмите кнопку New и в открывшемся окне Депозитария на странице New выберите пиктограмму **Application**. В окне Менеджера Проектов появится вершина, соответствующая создаваемому вами тестовому приложению
13. Выполните команду File | Save Project As и сохраните ваш проект под каким-то именем (например, **PTestDLL**).
14. Выделите в окне Менеджера Проектов вершину группы, щелкните правой кнопкой мыши и из всплывшего меню выберите команду Save Project Group As. Сохраните группу, например, под именем **TestDLL**.

Рис. 7.14

Окно Менеджера Проектов с загруженной DLL и тестом



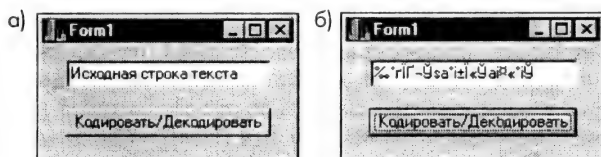
15. Разместите на форме окно редактирования и кнопку (рис. 7.15). В обработчик щелчка на кнопке поместите, например, такой оператор:

```
Edit1->Text = Code_Dec(Edit1->Text.c_str(), 'A');
```

Он берет текст, занесенный пользователем в окно редактирования **Edit1**, кодирует его с помощью нашей функции **Code_Dec** и возвращает закодированную строку в **Edit1**.

Рис. 7.15

Тестовое приложение DLL: исходный текст в окне (а) и результат кодировки (б)



16. Включите в модуль после директивы препроцессора **#pragma hdrstop** директиву, подключающую заголовочный файл библиотеки:

```
#include "MyDLL.h"
```

17. Осталось подключить к тестирующему приложению файл **.lib**, обеспечивающий статическое присоединение библиотеки. Для этого прежде всего активизируйте в окне Менеджера Проектов ваше приложение. Это можно сделать

двумя способами: двойным щелчком на вершине тестирующего проекта, или выделением этой вершины и нажатием кнопки *Activate* (см. рис. 7.14). Затем щелкните правой кнопкой мыши и выберите из всплывшего меню команду *Add*. Во всплывшем диалоговом окне выберите шаблон файлов «*Library file (.lib)*» и выберите файл **MyDLL.lib**. Это обеспечит при запуске вашего приложения статическое присоединение библиотеки. Вы можете посмотреть, как это делается, выполнив команду *View | Project Source*. В появившемся в окне Редактора Кода головном файле проекта вы увидите оператор

```
USELIB("MyDLL.lib");
```

Именно этот оператор и обеспечивает статическое присоединение библиотеки **MyDLL**.

Теперь все завершено. Вы можете сохранить проект, откомпилировать и выполнить его. Результаты работы этого приложения были показаны на рис. 7.15. Занесите в окно редактирования какой-то текст и нажмите кнопку. В окне появится абракадабра (рис. 7.15 б) — это закодированная, зашифрованная исходная строка. Нажмите кнопку повторно. В окне опять появится исходный текст (рис. 7.15 а) — результат декодирования (расшифровки) текста.

7.6 Пакеты

7.6.1 Общее описание концепции пакетов

Вы уже использовали пакеты в предыдущих разделах, создавая новые компоненты. Но теперь давайте рассмотрим этот инструмент более подробно.

Одним из основных достоинств C++Builder является то, что в результате проектирования создаются автономные выполняемые файлы **.exe**, т.е. приложение и все его ресурсы размещаются в одном выполняемом файле. Причем по сравнению с рядом других систем, позволяющих создавать автономные модули, размеры файлов в C++Builder достаточно небольшие. Автономный файл не требует наличия на компьютере пользователя не только среды C++Builder, но и каких-либо специальных библиотек C++Builder. Так что при разработке отдельного проекта, конечно, целесообразно создавать такой автономный выполняемый файл.

Однако, если у вас создано много выполняемых файлов или если вам надо передавать их по сети множеству пользователей, то размеры файлов становятся существенным критерием разработки. Стремление уменьшить затраты на хранение и распространение выполняемых файлов привело фирму Borland к концепции пакетов.

Пакеты (**Packages**) — это специальные динамически присоединяемые библиотеки **DLL**, содержащие библиотеки визуальных компонентов и другие объекты, функции, процедуры и т.д. Эти **DLL** позволяют вам создавать очень небольшие выполняемые модули, обращающиеся за поддержкой к пакетам. Вы можете также скомпилировать в пакеты свои собственные компоненты и библиотеки. Файлы пакетов имеют расширение **.bpl** (**Borland package library**), чтобы отличать их от обычных **DLL**.

Пакеты разделяются на два типа: *пакеты времени проектирования* и *пакеты времени выполнения*.

Пакеты времени проектирования C++Builder вызывает в процессе проектирования. Эти пакеты используются для установки компонентов в палитре компонентов среды разработки C++Builder или для создания специализированного редактора свойств для заказных компонентов. Пакеты времени проектирования не используются приложениями; они используются только самой средой C++Builder. Так что на этих пакетах далее мы останавливаться не будем.

Пакеты времени выполнения содержат библиотеки визуальных компонентов C++Builder. Ниже будет рассказано, как вы можете узнать состав отдельных пакетов и понять, какие пакеты используются в вашем приложении, а какие нет.

Пакеты времени выполнения содержат библиотеки визуальных компонентов C++Builder. Вы можете добавить к C++Builder заказные пакеты, разработанные вами или где-то приобретенные. При использовании пакетов времени выполнения вы должны передавать пользователю не только ваш выполняемый модуль, но и все пакеты времени выполнения, которые используются им. За счет того, что большая часть кодов помещается в этих пакетах, размеры ваших выполняемых модулей существенно сокращаются. Например, модуль в 350 КБ может быть сокращен примерно до 22 КБ. Вы передаете пользователям один раз все пакеты и устанавливаете их на компьютерах клиентов. А затем, когда вы делаете новые приложения, вам достаточно передавать только небольшие исполняемые модули.

Так что у вас есть две альтернативы:

- создавать законченные, автономные выполняемые модули
- использовать поддержку пакетов времени выполнения

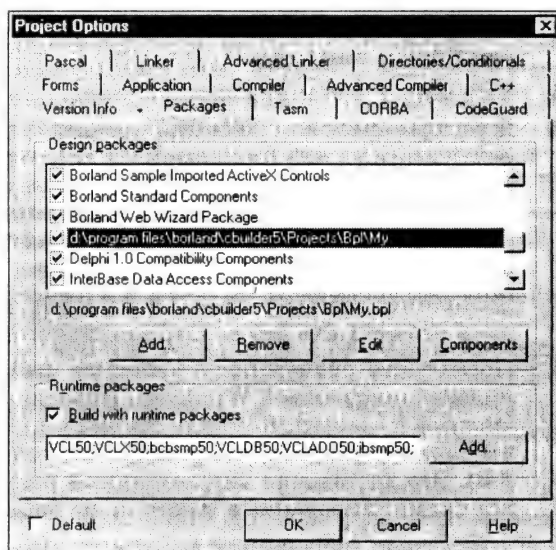
Отдельно следует сказать об использовании пакетов времени выполнения в процессе разработки приложения. Значительный выигрыш в размерах файлов делает целесообразным в процессе разработки и отладки приложения использовать режим поддержки пакетов, чтобы не иметь дело с большими файлами. А при заключительной компиляции проекта можно принимать решение: допустимо ли по условиям эксплуатации вашего приложения использовать поддержку пакетов, или надо компилировать автономный модуль.

7.6.2 Поддержка пакетов

Чтобы использовать пакеты в вашем приложении, вы должны сначала обратиться к поддержке пакетов, а затем компилировать и строить свое приложение. Поддержка пакетов определяется настройками соответствующих опций проекта. Для установки этих опций выполните команду Project | Options и в появившемся диалоговом окне опций проекта Project Options выберите страницу Packages — пакеты. Ее вид показан на рис. 7.16.

Рис. 7.16

Страница Packages окна опций проекта



В верхнем окне *Design packages* вы можете увидеть список используемых пакетов времени проектирования. В частности, вы видите на рис. 7.16 строку пакета, созданного вами при проектировании нового компонента в разделе 7.3 (если, конечно, вы его создавали). Вы можете добавить в список новые пакеты (кнопка *Add*) или удалить пакет (кнопка *Remove* или щелчок на окне индикатора рядом с выделенным пакетом). Впрочем, удалять системные пакеты вряд ли стоит. Лишние пакеты никак не повлияют на ваше приложение. Но зато, когда вы захотите создать новое приложение с компонентами, хранившимися в удаленном пакете, вам это не удастся сделать. Впрочем, в этом случае вы можете вновь установить требуемый пакет кнопкой *Add*.

Кнопка *Edit* (редактировать) доступна только если в окне *Design packages* выделен пакет, созданный вами. Для системных пакетов эта кнопка не доступна.

Кнопка *Components* позволяет вам просмотреть список компонентов, хранящихся в выделенном вами пакете.

Индикатор *Built with runtime packages* устанавливает режим поддержки пакетов. Если этот индикатор установлен, то выполняемый модуль вашего приложения будет требовать поддержки пакетов времени выполнения. При выключенном индикаторе *Built with runtime packages* вы создадите полностью автономный выполняемый модуль.

Индикатор *Default* позволяет принять установленную вами конфигурацию пакетов по умолчанию для всех новых проектов.

При изменениях списка в окне *Design packages* одновременно будет меняться и перечень пакетов времени выполнения *Runtime packages* в окне редактирования внизу. Для каждого конкретного приложения вы можете удалить из списка *Design packages* очень многие пакеты времени проектирования. Но никакого выигрыша в размере выполняемого модуля вы за счет этого не получите, поскольку *C++Builder* в любом случае не включит в модуль ничего лишнего. Однако, удаляя из этого списка все лишнее и повторяя создание выполняемого модуля вы можете установить тот минимальный набор пакетов, который вы должны поставлять вместе с вашим приложением (конечно в случае, если вы используете поддержку пакетов).

Чтобы почувствовать, что дают пакеты времени выполнения, проведите небольшой эксперимент. Постройте простейшее приложение *C++Builder* с пустой формой:

1. Откройте новый проект. Сохраните его под именами по умолчанию **project1.dpr** и **unit1.cpp**.
2. Выполните команду **Project | Options**.
3. В раскрывшемся окне опций проекта **Project Options** выберите страницу **Linker** и выключите на ней индикатор **Use dynamic RTL** (использовать динамическое связывание). Перейдите на страницу **Packages** и выключите на ней индикатор **Built with runtime packages** (строить с пакетами времени выполнения). Перейдите на страницу **Compiler** и нажмите кнопку **Release**, обеспечивающую создание файла без отладочной информации (см. раздел 14.2.9.2 главы 14).
4. Выполните команду **Project | Build**. В результате будет создан выполняемый модуль **project1.exe** без поддержки пакетов.
5. Посмотрите размер получившегося файла, воспользовавшись для этого, например, программой **Windows «Проводник»** или выполните команду **Project | Information for project**. Команда открывает окно информации о проекте, в котором вы сможете увидеть, что размер вашего выполняемого файла 359936 байт (351 КБ). Это достаточно внушительный размер, поскольку файл включает в себя все скомпилированные в него коды библиотеки компонентов.

6. Опять выполните команду Project | Options, и в окне опций проекта Project Options на странице Linker включите индикатор Use dynamic RTL, а на странице Packages включите индикатор Built with runtime packages.
7. Перестройте ваш выполняемый модуль, выполнив команду Project | Build (как в пункте 4).
8. Посмотрите теперь размер вашего файла **project1.exe** с поддержкой пакетов. Размер сократился до 22528 байт (22 Кб).

Как видите, выигрыш весьма значительный: без поддержки пакетов размер файла составляет 359936 байт (351 КБ), а с поддержкой — 22528 байт (22 Кб), т.е. примерно в 16 раз меньше. Вроде бы прекрасно! Но учтите, что прежде, чем пользователь сможет на своем компьютере применить ваше приложение, на этом компьютере надо поставить используемые приложением пакеты.

Узнать, какие пакеты и DLL нужны для распространения вашего приложения можно с помощью программы **tdump.exe**, поставляемой вместе с C++Builder и находящейся в каталоге ...\\bin. Эта программа работает в DOS. Чтобы воспользоваться ею, перейдите в DOS в каталог, в котором расположен файл вашего приложения **Project1**, и выполните, следующую команду:

```
tdump Project1.exe > dump.txt
```

В результате **tdump** проанализирует ваш файл и занесет результаты анализа в указанный вами текстовый файл **dump.txt**. Посмотрев этот файл, вы найдете в нем, наряду с прочими сведениями, строки вида

```
Imports from VCL50.BPL
  __fastcall System::initialization()
...
Imports from BORLNDMM.DLL
  (ord. = 2)
Imports from KERNEL32.DLL
  FreeLibrary
...
Imports from CC3250MT.DLL
  operator delete(void *)
...
```

Таким образом вы получите полный список пакетов и DLL, используемых вашим приложением. Эти пакеты и библиотеки должны быть на компьютере пользователя, чтобы он мог работать с вашим приложением. Впрочем, импортируемые DLL обычно типичны для Windows и имеются на любом компьютере. К тому же их объем, как правило, невелик. А файл пакета VCL50.BPL, который, как видно из приведенного выше текста, требуется для работы приложения, имеет размер 2023424 байта (1976 КБ или 1.93 МБ). Это, конечно, внушительный размер. Но дело в том, что этот файл вам надо поставить пользователю один раз и он обеспечит поддержку подавляющего большинства ваших приложений, файлы которых будут небольшими.

Давайте прикинем, когда это становится выгодным. Предположим, что средний размер файла приложения с поддержкой пакетов времени выполнения — 25 КБ, а без поддержки в 16 раз больше — 400 КБ. Тогда, если пользователь использует в своей работе N приложений, то без поддержки пакетов ему потребуется для хранения файлов N*400 КБ дискового пространства, а при поддержке пакета VCL50 — (1976 + N*25) КБ. Нетрудно посчитать, что уже при N = 6 поддержка пакетов становится выгодной.

Таким образом, в каждом конкретном случае надо решать, создавать ли вам автономный выполняемый модуль, или использовать поддержку пакетов времени выполнения. Очевидно, что при создании отдельной прикладной программы лучше делать автономный модуль. А при создании большой серии приложений, кото-

рые будут использоваться одними и теми же пользователями, возможно вариант с поддержкой пакетов может быть предпочтительнее.

Использовать, или не использовать поддержку пакетов времени выполнения — вопрос, возникающий только при заключительной компиляции уже готового приложения. Сам процесс проектирования никак не изменяется от того, будете или не будете вы использовать поддержку пакетов. А поскольку поддержка пакетов позволяет существенно уменьшить затраты дискового пространства под хранение выполняемых модулей, то можно дать следующий совет.

Совет

Для экономии места на диске при параллельной разработке нескольких проектов или нескольких вариантов одного проекта имеет смысл в процессе разработки включить поддержку пакетов времени выполнения. Это позволит вам более чем на порядок сократить размеры генерируемых выполняемых модулей.

Для использования этой возможности выполните команду Project | Options. В раскрывшемся окне опций проекта выберите страницу Packages и включите на ней индикатор Built with runtime packages. Одновременно полезно включить в том же окне (рис. 4.1) индикатор Default, что обеспечит статус этой установки как установки по умолчанию для всех ваших будущих проектов.

Только не забудьте при заключительной компиляции законченного проекта выключить опцию поддержки пакетов, если вы намерены передавать пользователям автономный выполняемый модуль.

Глава 8

Разработка справочной системы (создание файлов .hlp)

8.1 Проектирование справочной системы

Разработка справочной системы осуществляется не C++Builder, а средствами Windows. Просмотр ее также осуществляется программой Windows WinHelp. C++Builder только позволяет встроить справочную систему в приложение (см. раздел 4.1.9). Однако, поскольку полноценное приложение немислимо без хорошо разработанной справочной системы, мы коротко рассмотрим некоторые основные вопросы, связанные с ее созданием.

Разработка справки состоит из двух основных этапов:

- Создание файла или нескольких файлов, содержащих темы справок. Это делается с помощью любого текстового редактора, например, с помощью Microsoft Word.
- Компиляция справки в один или несколько файлов .hlp и отладка всей справочной системы. Это осуществляется с помощью специальных программ: HCRTF — Microsoft Help Workshop для Windows 95/98/2000 и NT и программ HC31 или HCP для Windows 3.x. При этом создаются вспомогательные файлы проекта и некоторые другие.

Рассмотрим указанные этапы работы. В основном будет рассмотрена методика создания справок для Windows 95/98/2000 и NT с помощью такого инструмента, как Microsoft Help Workshop 4. Соответствующие программы и файлы, необходимые для этого, обычно расположены в C++Builder каталоге .../Help/Tools. Особенности справок для Windows 3.x рассмотрены в разделе 8.4.

Прежде, чем создавать тексты справок, надо продумать систему в целом. Надо решить, как должно выглядеть основное окно, какую информацию выносить в дополнительные окна, что выносить во всплывающие окна, как должен выглядеть и какие разделы включать предметный указатель и окно содержания (это окно в виде дерева, содержащего изображения книг и документов каждый из вас видел, работая с какими-нибудь справками). В отношении основного окна надо решить, какие кнопки в своем заголовке оно должно иметь, нужна ли начальная часть, не подающаяся прокрутке, и что в нее должно входить, продумать общий стиль справки. При решении вопроса о дополнительных окнах и распределении функций между основным и дополнительными окнами надо иметь в виду, что дополнительное окно не имеет полосы меню и не обеспечивает доступ к истории и содержанию. С другой стороны у основного окна тоже имеется ограничение: оно не может быть сделано автоматически подстраивающим свои размеры под объем текста в нем. Не останавливаясь на подробностях процесса составления проекта справки, следует сказать, что этот процесс, конечно, трудно формализовать и любой первоначальный проект подвергнется переделкам в процессе его реализации. Но все-таки помните, что от удачного или неудачного планирования справки во многом зависит удобство пользователя, а, значит, и его отношение к вашему приложению, для которого готовится справка.

8.2 Создание файла тем справок

8.2.1 Написание текстов тем

Файл тем справок создается с помощью текстового редактора, например, с помощью Microsoft Word. Каждая тема или кадр (в дальнейшем он будет отображаться в отдельном окне) занимает одну страницу. Друг от друга темы отделяются символом разрыва страницы. В Word это осуществляется или командой Вставка | Разрыв и выбором в диалоговом окне опции Начать Новую страницу, или нажатием клавиш Ctrl-Enter.

Порядок расположения тем в файле безразличен, кроме одной темы — Содержание, которая по умолчанию располагается в файле первой и содержит ссылки на другие темы. На этот кадр передает управление WinHelp при щелчке пользователя на кнопке Содержание. Впрочем, если вы включаете в свой проект специальный файл Содержания, то именно этот файл будет определять, что увидит пользователь при щелчке на кнопке Содержание.

При написании темы можно использовать многие возможности Word: выбор атрибутов шрифта (полужирный, курсив), цвет шрифта, табуляцию, подчеркивание, включение в текст таблиц (правда, без сетки и без разноцветной заливки) и т.п. Шрифты тоже можно использовать различные. Но увлекаться этим не стоит, так как если у пользователя на компьютере не окажется соответствующего шрифта, он не сможет прочесть ваших текстов. Так что лучше всего использовать обычные системные шрифты (для Windows 95/98 и NT это MS Sans Serif).

В кадр могут специальным образом включаться рисунки, кнопки и т.д.

Каждая тема содержит ряд рассмотренных ниже сносков, определяющих ее отображение в WinHelp.

Темы могут содержать так называемые горячие области (hotspot): выделенные слова или кнопки, позволяющие пользователю переходить из данной темы в другие (позднее мы рассмотрим, как организуются эти выделения). При этом существует несколько возможностей переходов: прямой переход на заданную тему, переход с помощью макроса **KLink**, который может предложить пользователю выбор из тех тем, в **К**-сносках которых встречаются заданные ключевые слова, и с помощью макроса **ALink**, практически идентичного **KLink**, но сравнивающего ключевые слова с другим видом сносков — **А**-сносками.

По умолчанию строки текста при их отображении в окне справки будут «заворачиваться», т.е. та часть строки, которая не помещается в слишком узком для нее окне, переносится на новую строку. Это удобно, поскольку позволяет пользователю свободно изменять размеры окна, не затрудняя себе чтение текста. Однако, в некоторых случаях это может оказаться нежелательно, например, при отображении каких-то таблиц. Можно указать справочной системе, что она не должна заворачивать какие-то строки текста. Для этого надо выделить в Word эти строки, выполнить команду Формат | Абзац и в открывшемся диалоговом окне на странице Положение на странице установить опцию Не разрывать абзац. Тогда, если ширины окна справки не хватает, чтобы отобразить всю длину отмеченных таким образом строк, в окне появится полоса горизонтальной прокрутки, но строки разрываться и переноситься не будут.

Если в теме много строк, то для их просмотра пользователь будет пользоваться вертикальной прокруткой. Однако, часто желательно указать какую-то область в начале текста темы, которая оставалась бы «замороженной» и не прокручивалась. Например, это может быть заголовок темы, шапка таблицы и т.п. Для того, чтобы указать область, не подвластную прокрутке, надо выделить ее в Word, выполнить команду Формат | Абзац и в открывшемся диалоговом окне на странице Положение на странице установить опцию Не отрывать от следующего.

В тему можно **добавлять изображения** в виде файлов типов **.bmp**, **.dib**, **.wmf**, **.shg**, **.mrh**. Для этого можно просто воспользоваться буфером обмена, занеся туда изображение из какой-нибудь графической программы и прочитав ее в Word командой Правка | Вставить. Так целесообразно поступить, если данное изображение используется только в одном месте файла текстов тем. Если же оно используется в нескольких местах, то экономичнее использовать команды

```
{bmc <имя файла>},  
{bml <имя файла>}
```

или

```
{bmr <имя файла>}.
```

Первая из них позиционирует изображение так же, как обычные символы. Последующий текст продолжается в той же строке. Вторая — позиционирует изображение у левого края страницы и текст обтекает его справа. Третья — позиционирует изображение у правого края страницы и текст обтекает его слева. Например, вы можете написать такой текст:

Пиктограмма {bmc pict1.bmp} на инструментальной панели
соответствует команде ...

В том месте, где вы написали **{bmc pict1.bmp}**, появится изображение, содержащееся в файле «pict1.bmp», и сразу за изображением последует продолжение текста.

Вы можете добавить в текст **кнопки**, нажимая которые пользователь будет запускать тот или иной *макрос*. Это делается командой

```
{button <надпись>, <список макросов>}
```

В этой команде **<надпись>** — это та надпись, которая появится на изображении кнопки, а **<список макросов>** — перечень макросов, которые должны выполняться при нажатии кнопки. Если макросов несколько — они разделяются двоеточиями. Пример описания кнопки:

```
{button см. также, KLink(Меню)}
```

Это приведет к тому, что при работе со справкой в этом месте появится кнопка с надписью «см. также». Если пользователь нажмет эту кнопку, то увидит окно Найденные разделы, в котором ему будет показан список тем, содержащих в своих К-сносках (см. об этом ниже) название «Меню».

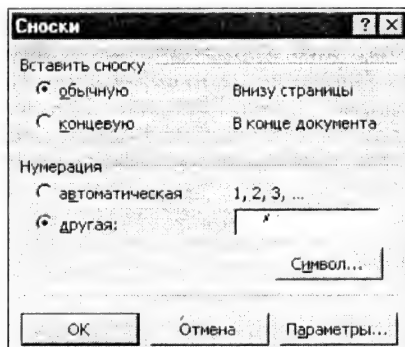
Файл тем справок сохраняется в файле формата RTF — обогащенном текстовом формате. Для сохранения его в таком виде надо в Microsoft Word выполнить команду Файл | Сохранить как и в открывшемся диалоговом окне установить опцию Тип файла равной Текст в формате RTF. После этого надо нажать кнопку Сохранить. Файл сохранится с расширением по умолчанию **.rtf**. Все это надо проделать только при первом сохранении файла. В дальнейшем он будет сохраняться в том же формате автоматически при выборе просто команды Сохранить.

8.2.2 Сноски

Каждая тема снабжается сносками, определяющими ее наименование и ряд других свойств. Сноски в Word делаются командой Вставка | Сноска, после чего открывается диалоговое окно Сноски (см. рис. 8.1). В нижнем его окне следует указать символ, используемый в сноске. Какие именно символы надо использовать в тех или иных сносках, будет рассмотрено ниже. Все сноски помещаются перед текстом темы, т.е. в первых позициях кадра. Чтобы наблюдать и редактировать тексты сносок Word должен быть переключен в режим Разметка страницы.

Рис. 8.1

Вставка сноски в текст файла тем в Word



8.2.2.1 Сноска

Сноска # обозначает уникальный *идентификатор темы*, по которому на нее могут ссылаться другие темы. Этому идентификатору в дальнейшем будет ставиться в соответствие номер, по которому на данную тему может ссылаться использующее справку приложение. Так что любой кадр должен снабжаться меткой #. Идентификатор, указываемый как текст этой ссылки, имеет чисто служебное назначение, пользователь его нигде не видит. Идентификатор может писаться латинскими или русскими буквами и состоять из одного или нескольких слов. Примеры таких сноска: #Main, #Содержание, #О программе.

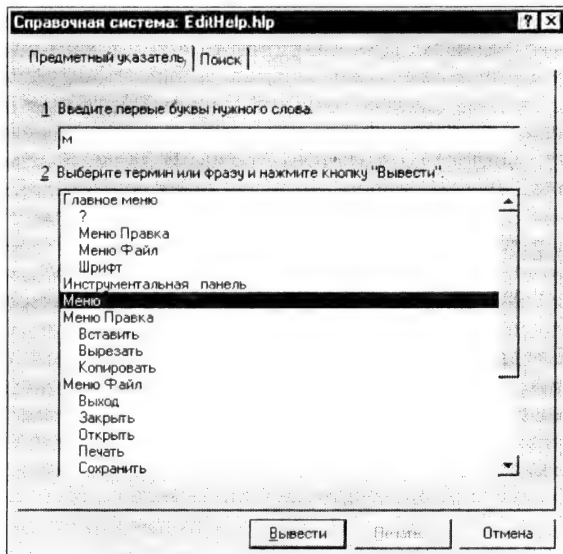
8.2.2.2 Сноска К

Сноска с символом **К** (заглавная латинская буква) должна включаться в кадр, если надо, чтобы тематика этого кадра отображалась при работе со справкой в окне справочной системы на странице «Предметный указатель» (см. рис. 8.2) в списке, из которого пользователь может выбрать требуемую тему по ее первым буквам или просто пролистав список. Те названия тем, которые пользователь видит в предметном указателе — это и есть тексты сноска **К** соответствующих кадров.

Текст сноски может состоять из одного или нескольких слов. Например, ^КМеню Правка. В этом случае при работе со справкой в списке указателей появится строчка «Меню Правка», по которой пользователь может выйти на данную тему.

Рис. 8.2

Страница «Предметный указатель» справочной системы



Можно также ввести несколько различных обозначений для одного и того же кадра, разделяя их точками с запятой. Например, ссылка

^кМеню правка; Меню

обеспечит две строки в указателе: «Меню правка» и «Меню», которые будут ссылаться на одну и ту же тему. Пользователь сможет выйти на нее по любой из этих строчек.

Можно обеспечить и двухуровневые ссылки, которые видны на рис. 8.2. Для этого пишется название темы первого уровня, затем ставится запятая и пишется название темы второго уровня. Причем название темы первого уровня может соответствовать данному кадру, а может относиться и к другому кадру. Например, текст сноски

^кМеню Правка; Меню Правка, Вырезать; Меню Правка, Копировать;
Меню Правка, Вставить; Меню; Главное Меню, Меню Правка

приведет к тому, что в окне указателя появятся строки (см. рис. 8.2)

Меню Правка
Вырезать
Копировать
Вставить

Первая из этих строк определена в первом элементе сноски. Следующие три строки второго уровня определены во втором, третьем и четвертом элементах сноски. Пятый элемент сноски определяет, что к данной теме пользователь может обратиться и по строчке «Меню». Наконец, последний элемент сноски определяет, что к названию другого кадра — «Главное Меню» добавится строчка второго уровня «Меню Правка». И по всем этим строчкам пользователь сможет выйти на данный кадр.

Элементы, указанные в сносках **К**, используются не только в списке указателя, но и при организации переходов между темами по ключевым словам с помощью макроса **KLink** (см. раздел 8.2.3.2).

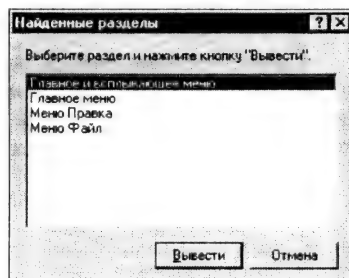
8.2.2.3 Сноска \$

Сноска с символом '\$' определяет заголовок данной темы. Этот заголовок используется в WinHelp в ряде режимов работы, в частности, в системе Поиск, в окне История, позволяющем пользователю вернуться к одной из уже просмотренных тем, и в ряде других режимов работы. Этот заголовок используется также во вспомогательном окне указателя тем в случаях, если какая-то строка основного указателя относится сразу к нескольким темам. Поясним это.

Выше были рассмотрены сноски **К**. Ничто не мешает в этих сносках для разных кадров указать одинаковый элемент. Это облегчает пользователю поиск нужной информации. Например, если вы описываете темы, связанные с главным меню приложения и его разделами Файл, Правка и др., вы можете во всех этих темах в сноски **К** внести элемент «Меню». Тогда строка «Меню» в предметном указателе будет относиться сразу к нескольким темам и пользователь, выбравший эту строку, должен иметь возможность уточнить свой выбор, указав одну из конкретных

Рис. 8.3

Окно «Найденные разделы» справочной системы



тем. Для этого система справки автоматически предъявит пользователю окно «Найденные разделы», вид которого представлен на рис. 8.3. Строки, обозначающие в этом окне конкретные темы задаются в кадрах с помощью сносок с символом \$. Таким образом, сноски \$ имеет смысл включать только в те кадры, которые в своих сносках **K** имеют элементы, используемые также в сносках **K** других кадров.

8.2.2.4 Сноска A

Сноска **A** (заглавная латинская буква) аналогична по синтаксису сноске **K**. Но ее тексты не включаются в указатель, как для сноски **K**, а используются только для переходов по ключевым словам с помощью макроса **ALink**. Таким образом, если тексты сносок **K** участвуют в двух различных процессах — поиске по указателю и поиске по ключевым словам, то сноски **A** позволяют развязать эти два процесса.

Хороший стиль программирования

Включайте в ваши темы сноски **A**. Они позволяют развязать процессы поиска по указателям и по ключевым словам. Это позволяет вам задавать в сносках **K** развернутые пояснения, удобные пользователю при работе с предметным указателем.

8.2.2.5 Сноска +

Сноска с символом '+' используется для указания последовательности просмотра тем с помощью кнопок «>>» и «<<». Эти сноски имеют смысл только в случаях, когда вы включаете в окно справки соответствующие кнопки просмотра (кнопки Browse, включаются в файле Проекта — см. раздел 8.3.1). Если сноски + есть, но в них не указано никаких значений, то Help Workshop устанавливает автоматически последовательность просмотра в соответствии с последовательностью тем в вашем файле. В кадрах, в которых сносок + нет, кнопки просмотра не доступны. Тексты сносок могут быть просто номерами (рекомендуется обозначать их с предшествующими нулями: 001, 002 и т.д.) или идентификаторами.

8.2.2.6 Сноска !

Сноска с символом '!' используется для указания одного или нескольких макросов, которые должны сработать перед появлением окна данной темы на экране. Если указывается несколько макросов, то они разделяются запятыми или точками с запятой. Эти сноски используются только в достаточно сложных справках. Подробнее о макросах см. в разделе 8.2.4.

8.2.2.7 Сноска *

Сноска с символом '*' используется для указания связанных друг с другом тем, которые должны быть встроены в справку. Эти сноски позволяют, подготовив один файл тем, использовать его для генерации несколько разных файлов справок. В этом случае в файле проекта .hpj указывается, темы с какими значениями сносок * надо включить в справку — в файл .hlp. Темы без сносок * включаются в справку в любом случае.

8.2.2.8 Сноска >

Сноска с символом '>' используется для указания идентификатора окна, в котором должна отображаться тема. Окно с этим именем должно быть определено в файле проекта .hpj. Сноска > используется при передачах управления на данную тему из окна указателя или с помощью макросов **KLink** и **ALink**.

8.2.3 Переходы

В темы можно вводить переходы на другие темы несколькими изложенными ниже способами.

8.2.3.1 Непосредственные переходы

Для того, чтобы выделить в тексте темы некоторое слово или сочетание слов, при щелчке на котором пользователь перейдет на другую тему, надо сделать следующее. Сразу после этих слов (без пробела) надо написать идентификатор темы, на которую надо перейти. Затем соответствующее слово или сочетание слов выделяется двойным подчеркиванием. Это подчеркивание не будет видно пользователю при работе со справкой — для него оно заменится выделением цветом (обычно зеленым). Для того, чтобы в Microsoft Word обеспечить двойное подчеркивание, надо выделить курсором требуемое слово или сочетание слов и выполнить команду **Формат | Шрифт**. В открывшемся окне на странице **Шрифт** надо установить опцию **Подчеркивание** в положение **Двойное** и нажать **ОК**.

После этого следующий за подчеркнутыми словами идентификатор темы перехода оформляется как скрытый (невидимый). Для этого выполняется та же команда **Формат | Шрифт**. В открывшемся окне на странице **Шрифт** в разделе **Эффекты** надо установить индикатор опции **Скрытый** и проследить, чтобы опция **Подчеркивание** имела значение «(нет)».

Для того, чтобы вы сами видели этот скрытый текст и могли бы его редактировать, надо выполнить в Word команду **Сервис | Параметры** и на странице **Вид** в разделе **Непечатаемые символы** установить индикатор **Скрытый текст**. Если вам захочется напечатать на принтере текст вашего файла, то дополнительно на странице **Печать** в разделе **Печатать** надо тоже установить индикатор **Скрытый текст**. Не забудьте все это продумать. Иначе вам будет очень трудно работать с вашим файлом.

Приведем пример вставки в текст переходов. Пусть вы написали тему с текстом

О ПРОГРАММЕ

Данная программа является примером простого текстового редактора. Она позволяет загрузить документ из файла в окно редактирования или создать в этом окне новый документ.

Управление работой ведется с помощью Главного меню и быстрых кнопок.

Вы хотите выделить слова «окно редактирования», чтобы при щелчке на них пользователь переходил к теме с описанием работы с этим окном. Пусть идентификатор темы, где описана работа с этим окном, — «Окно». Аналогично вы хотите выделить слова «Главного меню», чтобы после щелчка на них пользователь переходил к теме с именем «Главное меню», и слова «быстрых кнопок» для перехода на тему «Buttons». Тогда после описанных ранее выделений нужных словосочетаний, указания для них соответствующих тем и введения ссылок текст приобретет вид:

*\$К О ПРОГРАММЕ

Данная программа является примером простого текстового редактора. Она позволяет загрузить документ из файла в окно редактирования или создать в этом окне новый документ.

Управление работой ведется с помощью Главного меню и быстрых кнопок.

* О программе

\$ О программе

К О программе; О программе, Назначение программы; О программе, Управление программой

При работе со справкой скрытый текст не будет виден. Пользователь увидит только тот текст, который был до введения указателей переходов. В этом тексте будут выделены зеленым цветом и подчеркнуты слова «окно редактирования», «Главного меню» и «быстрых кнопок». При щелчке на них пользователь перейдет к соответствующим темам.

При задании переходов имеются также возможности сообщить справочной системе некоторую дополнительную информацию. Если желательно, чтобы выделенные слова отображались цветом, принятым по умолчанию для всего текста (т.е. не выделялись зеленым цветом), то перед ссылкой на тему ставится знак *. Например: «быстрых кнопок*Buttons». Если к тому же вы хотите, чтобы эти слова отображались не подчеркнутыми, то вместо символа * надо поставить символ %. Например: «быстрых кнопок%Buttons».

Если ваша справка состоит из нескольких файлов .hlp и вы хотите указать переход на тему другого файла, то после указания идентификатора темы надо поставить символ @, после которого указать имя файла, в котором расположена эта тема. Например: «быстрых кнопокButtons@file2.hlp».

Задавая переход, вы можете указать имя окна (из числа определенных вами в файле Проекта — см. раздел 8.3.1), в котором вы хотите отобразить тему. Для этого после идентификатора темы вы ставите символ больше (>), после которого пишете имя окна. Например, «быстрых кнопокButtons>W2».

Можно также отображать тему, на которую вы переходите, во всплывающем окне. Такие окна используются чаще всего для дополнительной информации или пояснений. В отличие от основного окна всплывающее окно появляется, не убирая с экрана окна вызвавшей его темы. При любом действии пользователя всплывшее окно исчезает и пользователь оказывается в предыдущем окне.

Указание на переход к теме, отображаемой во всплывающем окне, осуществляется точно так же, как было описано выше, с единственным отличием — выделяемое слово или сочетание слов подчеркивается не двойной, а одинарной чертой. Например, «быстрых кнопокButtons».

8.2.3.2 Переходы по ключевым словам

Имеется два макроса — KLink и ALink, которые позволяют переходить не к заранее заданной теме, а к темам, в соответствующих сносках которых имеются заданные ключевые слова. Если заданные слова встречаются в нескольких темах, то пользователю показывается окно «Найденные разделы» (рис. 8.3) и он может сам определиться, на какую тему ему уходить.

Хороший стиль программирования

В сложных справках лучше во многих случаях использовать макросы KLink и ALink (прежде всего ALink) вместо непосредственных переходов. Эти макросы позволяют пользователю при наличии нескольких родственных тем выбрать ту, которая его более интересует в данный момент).

Хороший стиль программирования

В темы полезно вводить ссылки вида «см. также», в которых с помощью макроса ALink перечислять ключевые слова, связанные с данной темой. Это облегчает пользователю комплексное изучение интересующего его вопроса.

Макросы KLink и ALink имеют одинаковый синтаксис и действуют одинаково. Оба они могут использоваться, в частности, вместо рассмотренных выше непосредственных ссылок на темы. Различие между этими макросами состоит в том, что первый из них ищет заданные ключевые слова в K-ссылках, а второй — в

А-ссылках. Приведем синтаксис вызова макроса **KLink** (синтаксис **ALink** аналогичен, только надо заменить **KLink** на **ALink**):

```
KLink("<список ключевых слов>", <тип>,
      "<идентификатор темы>", <имя окна>)
```

<список ключевых слов> представляет собой одно или несколько ключевых слов или словосочетаний, разделенных точками с запятой. Если хотя бы одно из этих словосочетаний содержит запятую, то весь список заключается в двойные кавычки. Поиск ведется в **К-ссылках** сначала по первому слову. Если нашлось несколько тем, то пользователю показывается окно Найденные разделы. Если же не нашлось ни одной темы, начинается поиск по второму ключевому слову и т.д.

Все остальные элементы вызова макроса, кроме списка ключевых слов, являются не обязательными. **<тип>** определяет реакцию на найденные или не найденные ключевые слова и может принимать одно или несколько (разделяемых пробелами) следующих значений:

Символическое	Численное	Описание
JUMP	1	Если найдена только одна тема, соответствующая ключевым словам, то на нее сразу осуществляется переход
TITLE	2	Если ключевое слово находится более чем в одном файле справки (при справке, состоящей из нескольких файлов), то в окне Найденные разделы после названия темы пишется имя файла так, как оно определено в файле .cnt (см. раздел 8.3.3).
TEST	4	Макрос возвращает величину, указывающую, нашлось, или нет хотя бы одно соответствие ключевым словам.

<идентификатор темы> определяет, что если не найдено соответствия ключевым словам, то появится всплывающее окно с текстом, содержащимся в теме, на которую указывает этот идентификатор. Если идентификатор не задан, то при безуспешном поиске появляется диалоговое окно с текстом «Дополнительные сведения отсутствуют. (141)». Если идентификатор темы относится к другому файлу, то после него надо написать символ @, а затем — имя файла.

<имя окна> задает окно для отображения. Если этот параметр не задан, то используется окно, заданное в кадре темы, а если оно и там не задано, то используется окно по умолчанию.

Приведем примеры использования рассмотренных макросов. Текст

```
Меню!KLink(Меню; Главное меню,Jump) программы позволяет выполнить все операции.
```

приведет к выделению слова «Меню», щелкнув на котором пользователь увидит или список тем, содержащих в своих **К-сносках** ключевые слова «Меню» и «Главное меню», или, если есть только одна такая темы, то сразу перейдет на нее. Если в том же тексте заменить обращение к макросу на

```
!KLink(Меню; Главное меню,Jump,,W1)
```

то будет то же самое, но тема отобразится в окне с именем **W1** (если такое окно определено в файле проекта).

Оператор

```
{button Меню, ALink(Меню; Главное меню,Jump)}
```

приведет к появлению в кадре кнопки с надписью «Меню», при нажатии на которую будет та же реакция, что и в рассмотренном в начале примере, но поиск будет проходить в А-ссылках.

8.2.4 Макросы

Имеется множество макросов, помимо описанных выше **KLink** и **ALink**, которые можно использовать при разработке справки. Они могут запускаться из «горячих областей» — выделенных словосочетаний, кнопок и т.д., или при открытии той или иной темы, или при открытии справки в целом. Рассмотрение этих макросов выходит за рамки данной книги. Все они хорошо описаны в файле справки **Hcw.hlp**, который может быть вызван непосредственно из Windows или из Microsoft Help Workshop. Большинство этих макросов позволяют работать с кнопками окна справки, с меню, создавать и уничтожать элементы списков и т.д. Рассмотрим коротко только некоторые из них.

Отметим прежде всего макросы, позволяющие оперировать с файлами внешних программ. Эти макросы, введенные в WinHelp 4.0:

Макрос	Описание
ControlPanel	Открывает заданный элемент (файл .cpl) программы «Контрольная Панель».
ExecFile	Запускает указанную программу или открывает файл и запускает связанную с ним программу.
FileExist	Проверяет наличие указанного файла на компьютере пользователя.
ShellExecute	Открывает, печатает или запускает файл или программу.
ShortCut	Запускает указанную программу, если она еще не запущена, или активизирует ее и передает ей сообщение WM_COMMAND.

Впрочем, и в более ранних версиях WinHelp, даже в Windows 3.x имелся макрос **ExecProgram**, позволяющий запускать внешние программы.

Не останавливаясь детально на синтаксисе макросов, описанном в справке по подготовке Help, рассмотрим несколько примеров.

Приведенный ниже оператор создает кнопку, при нажатии на которую запускается элемент «Контрольной Панели» Дата/время.

```
{button Дата/время, ControlPanel(Timedate)}
```

Следующий оператор создает кнопку, при нажатии на которую запускается программа Windows «Калькулятор».

```
{button Калькулятор, ExecFile(Calc.exe)}
```

Следующий оператор создает кнопку, при нажатии на которую открывается исходный файл справки, т.е. вызывается программа, связанная с этим файлом. Поскольку он записан в формате RTF, то обычно с ним связана программа Word.

```
{button Topics.rtf, ExecFile(Topics.rtf)}
```

Следующий оператор проверяет, есть ли на компьютере файл приложения **myapp.exe**. Если есть, то это приложение запускается. В противном случае осуществляется переход на тему с идентификатором **install**.

```
IfThenElse(FileExist(myapp.exe), ExecFile(myapp),  
            JumpId(install_my_app))
```

Ряд макросов связан с созданием кнопок в полосе кнопок заголовка окна справки или с созданием разделов меню, не предусмотренных по умолчанию в справочной системе. Эти макросы обычно выполняются при открытии справки и включаются в описанный в следующем разделе файл Проекта справки, хотя кнопки и разделы меню могут создаваться или уничтожаться и в отдельных темах справки. Приведем примеры таких макросов.

Макрос

```
BrowseButtons()
```

создает в полосе кнопок заголовка окна справки кнопки просмотра вперед и назад (кнопки >> и <<). Эти кнопки обычно имеет смысл включать только при использовании в темах сносков упорядочивания⁺.

Макрос **CreateButton** создает в полосе кнопок заголовка окна справки новую кнопку с указанным именем, соответствующую указанному макросу. Синтаксис макроса **CreateButton** следующий:

```
CreateButton("<идентификатор>", "<надпись>", "<макрос>")
```

Здесь **<идентификатор>** — внутренний идентификатор кнопки (произвольный), который можно использовать, если вы потом захотите, например, в каких-то темах удалить эту кнопку. **<надпись>** — это то, что будет написано на кнопке. **<макрос>** — тот макрос, который будет выполняться при щелчке на этой кнопке. Например, макрос

```
CreateButton("History", "&История", "History()")
```

создаст кнопку с надписью История, при щелчке на которой будет выполняться макрос **History()**, отображающий окно со списком до сорока последних тем, просмотренных пользователем. Из этого списка пользователь может выбрать тему, к которой он хочет вернуться.

В заключение приведем несколько макросов, позволяющих добавлять новые меню и разделы в полосу меню окна справки. Первый из этих макросов — **InsertMenu**, добавляющий новое меню. Его синтаксис:

```
InsertMenu("<идентификатор меню>", "<надпись>", <номер>)
```

Здесь **<идентификатор меню>** — внутренний идентификатор меню, который можно использовать при последующих ссылках, **<надпись>** — надпись раздела, **<номер>** — порядковый номер меню (меню нумеруются слева направо, начиная с 0).

Второй макрос — **AppendItem**, вставляющий раздел в конец указанного меню. Его синтаксис:

```
AppendItem("<идентификатор меню>",  
           "<идентификатор раздела>", "<надпись>", "<макрос>")
```

Здесь **<идентификатор меню>** и **<идентификатор раздела>** — внутренние идентификаторы, используемые в справке для последующих ссылок на меню и его раздел, **<надпись>** — надпись, которая появится в меню, **<макрос>** — макрос или макросы, которые должны выполняться при выборе пользователем данного раздела меню.

Приведем пример совместного использования этих макросов. Операторы

```
InsertMenu("mexit", "Выход", 5)  
AppendItem("mexit", "exit", "Выход", "exit()")
```

заносят в полосу меню окна справки меню Выход на 5-е место и заносят в него раздел Выход, в котором выполняется макрос **exit()**, осуществляющий выход из справки.

Макрос **InsertItem** позволяет вставить раздел в меню, указанное его идентификатором. Для предусмотренных по умолчанию меню окна справки используются следующие идентификаторы:

Меню	Идентификатор
Файл	mnu_file
Правка	mnu_edit
Параметры	mnu_options
Закладка	mnu_bookmark
?	mnu_help
Всплывающее меню	mnu_floating

Пример использования макроса **InsertItem**:

```
InsertItem("mnu_help", "How", "Как использовать справку", "HelpOn()", 0)
```

В этом примере в меню справки вставляется на первое место (индекс 0 — последний элемент макроса) раздел Как использовать справку, при выборе которого выполняется макрос **HelpOn()**, обеспечивающий переход на стандартный файл, поясняющий использование справки.

8.3 Компиляция и отладка

Компиляция и отладка справки, предназначенной для 32-разрядного Windows, производится с помощью программы Microsoft Help Workshop, запускаемой из файла **Hcrtf**, расположенного в каталоге CBuilder5\Help\Tools. Эта программа позволяет легко создать файл Проекта справки, без которого ее нельзя компилировать, а если нужно, то создать еще некоторые вспомогательные файлы, например, файл Содержания справки и ряд других. Далее программа позволяет скомпилировать файл справки и проверить его в работе. В принципе можно создавать файлы Проекта, Содержания и др. просто в любом текстовом редакторе. Но тогда надо детально изучить их синтаксис. А Help Workshop позволяет автоматизировать всю работу, не требуя знания синтаксиса, так как эта программа сама создает тексты файлов, отражающие действия разработчика.

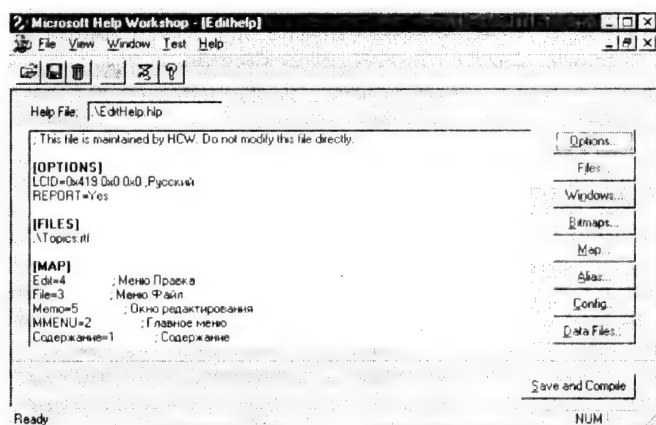
8.3.1 Создание файла Проекта справки

Для компиляции справки и ее связывания с использующим его приложением необходимо сформировать файл Проекта справки. Это текстовый файл в формате ASCII. Создание этого файла и отладка справки осуществляется программой Microsoft Help Workshop, которая распространяется вместе с C++Builder и расположена в каталоге ...CBuilder5\Help\Tools. Для создания нового файла Проекта, надо выполнить команду программы Microsoft Help Workshop File | New и в появившемся окне New выбрать опцию Help Project. Далее в окне Project File Name надо задать имя и каталог файла проекта справки. При этом учтите, что то же имя, какое вы зададите файлу Проекта, будет присвоено компилятором и завершающему файлу справки **.hlp** (впрочем, это можно отменить, задав другое имя файлу справки в окне Options на странице Files — это будет рассмотрено позднее). Стандартное расширение файла Проекта — **.hpj**.

В результате описанных действий перед вами появится окно заготовки файла Проекта (рис. 8.4), в котором первоначально будет занесен только раздел [Options]. Теперь, щелкая на соответствующих кнопках в правой части этого окна, можно создавать и заполнять другие разделы файла Проекта. В появляющихся при этом диалоговых окнах, описанных ниже (см., например, рис. 8.5) справа вверху имеется кнопка со знаком вопроса. Если нажать ее, возникает изображение вопроса-

Рис. 8.4

Окно редактирования файла
Проекта справки



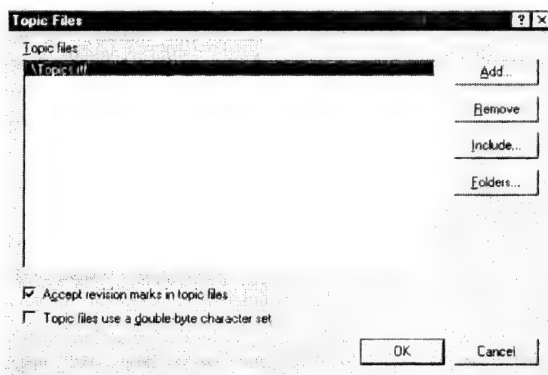
тельного знака, которое можно подвести к любому элементу окна, после чего всплывет пояснение функции этого элемента.

8.3.1.1 Кнопка Files

Кнопка Files в правой части окна рис. 8.4 (не путайте с аналогичным по названию разделом меню) позволяет указать имена файлов с текстами тем. Щелкните на этой кнопке. Перед вами появится окно «Topic Files» (рис. 8.5), в котором надо щелкнуть на кнопке Add и выбрать среди файлов подготовленный файл текстов справки. В результате в файле проекта появится раздел [Files] с внесенным в него именем файла (см. рис. 8.4). Если ваша справка компилируется из нескольких файлов тем, то аналогичным образом вы должны включить в раздел [Files] и остальные файлы.

Рис. 8.5

Окно «Topic Files» — ввод имен файлов тем



Для файлов тем, расположенных в том же каталоге, где располагается файл Проекта **hlpj**, имена файлов могут указываться без путей. Если же они расположены в другом каталоге, то они указываются с путем. Возможен и другой подход — в окне Topic Files (рис. 8.5) можно щелкнуть на кнопке Folders и выбрать папку или папки, в которых хранятся файлы тем. Тогда в файле Проекта появится оператор **Root**, в котором указана соответствующая папка.

Кнопка Remove в окне Topic Files позволяет удалить файл тем из списка. Кнопка Include позволяет включить в проект текстовый файл ASCII, содержащий список файлов тем.

Теперь перейдем к другим кнопкам окна Проекта, представленного на рис. 8.4.

8.3.1.2 Кнопка Windows

Кнопка Windows позволяет определить атрибуты окон, используемых в справке. При щелчке на этой кнопке открывается диалоговое окно «Window Properties» (свойства окон), представленное на рис. 8.6 а. Оно имеет ряд страниц. На странице General (общие характеристики) вы можете изменять список определенных вами типов окон, добавляя в него (кнопкой Add) или удаляя (кнопкой Remove) идентификаторы окон. Кнопка Include позволяет включить целый файл, содержащий список окон. Дополнительным окнам вы можете присваивать имена по вашему усмотрению. Эти имена затем могут использоваться в сносках > и в операторах переходов, о которых рассказывалось в разделе 8.2. Если же вы хотите определить характеристики основного окна, отказавшись от его характеристик по умолчанию, вы должны включить в список идентификатор **main**.

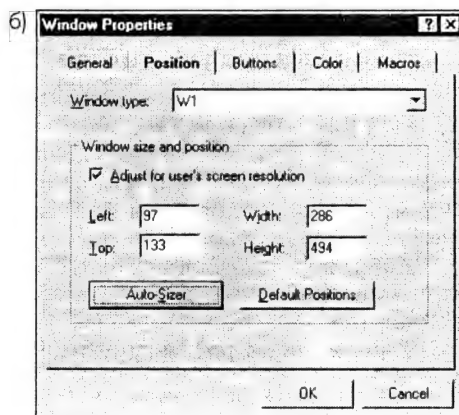
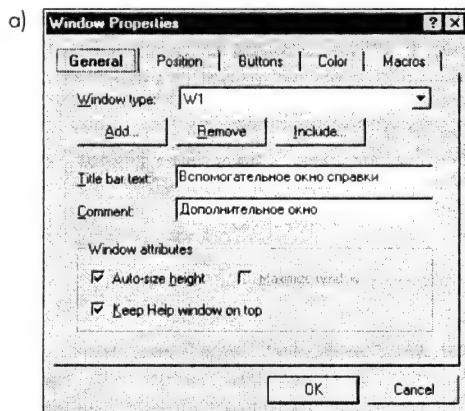
Расположенные внизу диалогового окна индикаторы позволяют установить опции, определяющие состояние окон. Установка опции Auto-size height (автоматическое изменение высоты) делает окно изменяющимся по высоте в зависимости от объема текста темы. Это удобно, если у вас есть темы, существенно различающиеся по объему. Главное окно не может содержать эту опцию. Опция Keep Help window on top позволяет обеспечить положение данного окна справки всегда поверх остальных окон.

Окно в середине страницы с заголовком Titlebar text позволяет задать текст, который будет записан в полосу заголовка окна справки. Если этот текст не задан ни тут, ни в файле Содержания .cnt, то в заголовке окна просто не будет никакого текста.

Страница Position (положение) диалогового окна Window Properties (рис. 8.6 б) позволяет выбрать положение и размеры окна. Здесь особенно полезной является

Рис. 8.6

Окно свойств окон справки: страницы General (а) и Position (б)



кнопка Auto-Sizer. Щелчок на ней вызывает появление на экране окна, представленного на рис. 8.7. Вы можете изменять размеры и позицию этого окна на экране. Когда вы расположите окно так, как вам хочется, щелкните на его кнопке ОК и выбранные вами положение и размеры передадутся окну вашей справки, атрибуты которого вы задаете.

Рис. 8.7

Задание размеров и местоположения окна



Остальные страницы диалогового окна Window Properties мы не будем подробно рассматривать, поскольку они вряд ли могут вызвать затруднения при работе с ними. Страница Buttons позволяет задать кнопки, которые должны быть в окне, страница Color позволяет выбрать цвета фона окна, причем отдельно для основной области текста, и для непрокручиваемого заголовка. Кнопка Macros дает возможность определить макрос, выполняемый в момент открытия окна.

При задании атрибутов главного окна (**main**) следует иметь в виду, что они не относятся ко входам в справку из приложения.

Предупреждение

Атрибуты главного окна (**main**), задаваемые в файле проекта, не действуют при входе в справку из приложения. Поэтому, во избежание неприятных накладок с различными стилями окон, надо для всех тем, в которые можно войти из приложения, добавить сноски >, в которых явным образом указать в качестве окна **main**.

8.3.1.3 Кнопка Bitmaps

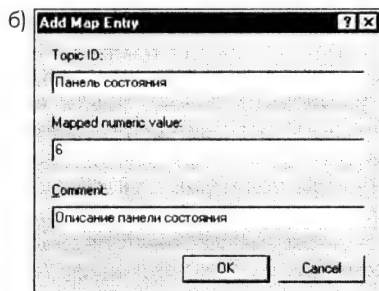
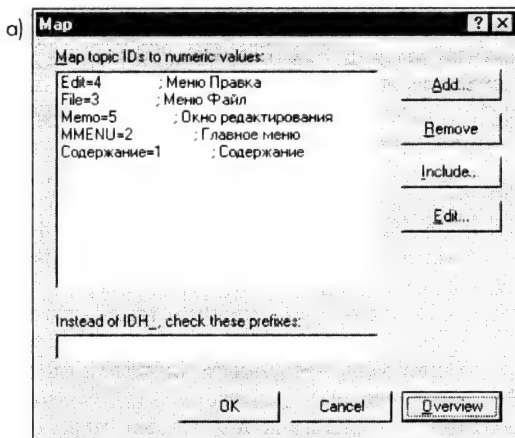
Кнопка Bitmaps позволяет указать папки, в которых Help Workshop сможет найти использованные в темах файлы изображений **.bmp**, если они расположены не в том каталоге, в котором находится файл проекта.

8.3.1.4 Кнопка Map

Если вы хотите, чтобы из вашего приложения можно было управлять файлом справки, необходимо создать таблицу соответствия номеров контекстной справки, задаваемых в приложении, и идентификаторов тем. Для этого надо щелкнуть на кнопке Map, в открывшемся диалоговом окне «Map» (рис. 8.8 а) щелкнуть на Add и в окне добавления нового входа Add Map Entry (см. рис. 8.8 б) ввести идентификатор темы (в окно Topic ID) и соответствующий номер (в окно Mapped numeric value), по которому будет осуществляться ссылка на эту тему из приложения. Можно также ввести комментарий в окно Comment. Этот комментарий просто будет занесен в файл проекта и никак не повлияет на справку. Но он очень пригодится вам, когда вы будете назначать значения **HelpContext** компонентам своего приложения. Затем надо щелкнуть на ОК, вернуться в окно «Map» и опять щелкнуть на ОК. В результате подобных действий в файле проекта появится раздел [MAP] (см. рис. 8.4).

Рис. 8.8

Задание соответствия номеров справки и идентификаторов тем: главное окно Map (а) и окно задания нового входа (б)



8.3.1.5 Кнопка Alias

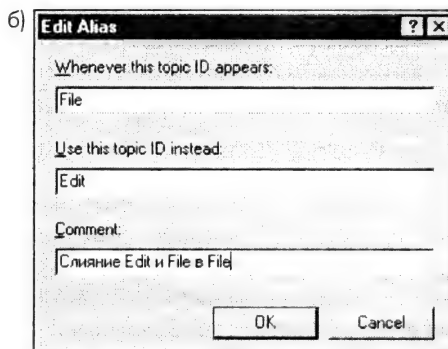
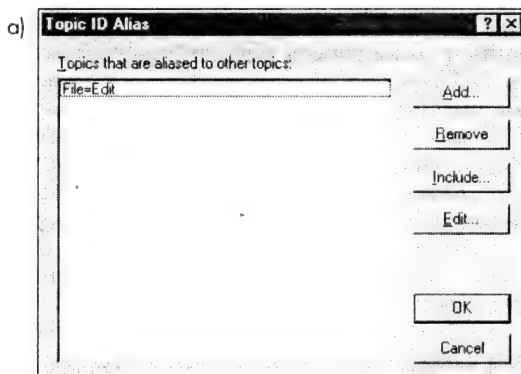
При нажатии на кнопку Alias (псевдонимы) появляется окно (рис. 8.9 а), подобное уже рассмотренным ранее, содержащее кнопки добавления, удаления, редактирования псевдонимов идентификаторов тем и включения сразу списка псевдонимов из файла. Псевдонимы могут вводиться, чтобы изменить переходы в файле справки или входы в него из приложения, не меняя текстов тем. Например, у вас сначала было две темы: одна из них с идентификатором File описывала меню «Файл» вашей программы, а вторая с идентификатором Edit описывала меню «Правка». Затем вы решили объединить оба описания в одной теме и дали этой теме идентификатор File, принадлежащий ранее первой теме. Можно, конечно, вручную просмотреть весь файл тем (или несколько файлов), найти все ссылки на Edit и заменить их на File. А можно установить идентичность этих двух ссылок с помощью псевдонима. Для этого нажимаете кнопку Alias, затем кнопку Add и в открывшемся диалоге (рис. 8.9 б) напишете в верхнем окне редактирования «File», а в нижнем «Edit». Тогда все ссылки и переходы на Edit заменятся на ссылки и переходы на File. В файле Проекта появится раздел [Alias], содержащий таблицу введенных псевдонимов.

8.3.1.6 Кнопка Config

При нажатии этой кнопки возникает диалоговое окно, подобное всем предыдущим, в котором можно указать макросы, которые должны выполняться при каждом входе в справку. Например, это может быть макрос **BrowseButtons()**, обеспечивающий появление в окне кнопок «>>» и «<<» для перемещения по темам вперед и назад, макрос **CreateButton**, создающий кнопку «История», или любые другие макросы (см. раздел 8.2.4).

Рис. 8.9

Задание псевдонимов: главное окно (а) и окно добавления псевдонима (б)



8.3.1.7 Кнопка Data Files

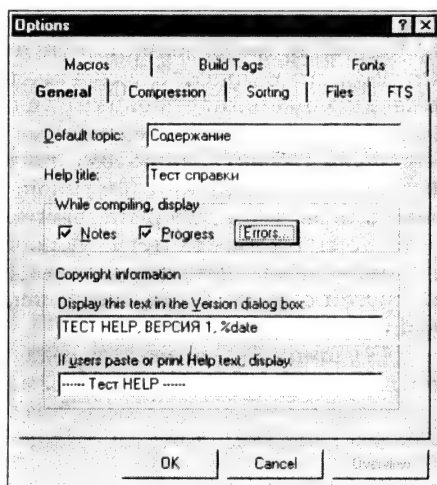
При нажатии этой кнопки возникает диалоговое окно, подобное предыдущим, в котором можно указать файлы данных для библиотек DLL, используемых в вашей справке.

8.3.1.8 Кнопка Options

Кнопка Options открывает многостраничное диалоговое окно, представленное на рис. 8.10. Страница General позволяет задать в окне Default topic идентификатор

Рис. 8.10

Задание темы по умолчанию

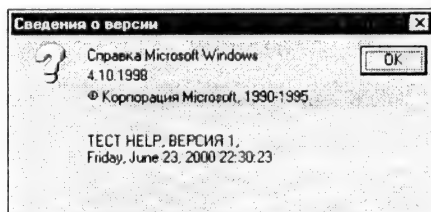


темы по умолчанию. Это тема, на которую будет осуществляться переход, при ошибочном задании идентификатора темы (после соответствующего предупреждения). Кроме того, если справка не имеет файла Содержания **.cnt**, то по этому идентификатору будет осуществляться переход при нажатии пользователем, работающим со справкой, кнопки Содержание (если тема в окне Default topic не задана, то в этом случае по умолчанию будет переход к первой теме первого файла). Опция Help title позволяет задать заголовок главного окна, если для проекта не определено окно **main** с соответствующим заголовком с помощью кнопки Windows.

Опция Display this text in the Version dialog box: позволяет задать текст, который пользователь будет видеть в диалоговом окне, когда при работе со справкой выберет в меню **?** раздел Версия. Пример этого окна приведен на рис. 8.11. В тексте можно задать ключевые символы «%date» и тогда в окно будет заноситься дата создания файла справки (см. рис. 8.11).

Рис. 8.11

Пример окна «О программе»



Опция If user paste or print Help text, display: позволяет задать текст, который будет присоединяться к тексту темы при его печати или при копировании в буфер обмена. Опции в средней части окна относятся к процессу компиляции справки и позволяют, в частности, с помощью кнопки Errors запретить появление некоторых сообщений об ошибках.

Страница Compression позволяет задать уровни сжатия при компиляции. Повышение уровня сжатия уменьшает размер результирующего файла, но увеличивает время компиляции.

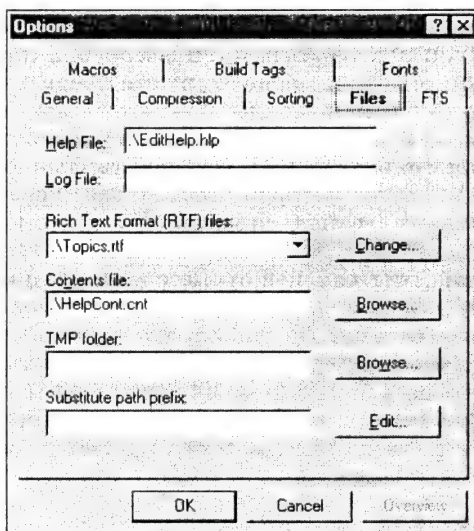
Страница Sorting определяет язык справки и сортировку в предметном указателе с игнорированием или с учетом различных обозначений (опция Non-spacing marks) и вспомогательных символов, таких, как пунктуация (опция Symbols).

Страница Files (рис. 8.12) в верхнем окне Help file позволяет задать имя скомпилированного файла справки **.hlp**, отличное от имени файла Проекта **.hpi**. В следующем окне, если хотите, можно задать имя файла протокола, в который будут заноситься сообщения о ходе компиляции. Следующее окно Rich Text Format (RTF) files просто дублирует информацию, которую вы задавали в окне Topic Files, нажав кнопку Files. Если вы создали для справки файл Содержания **.cnt**, то вы должны подключить его к проекту, задав его имя в окне Contents file. В следующем окне TMP folder вы можете задать папку для создаваемых в процессе компиляции временных файлов. По умолчанию временные файлы создаются в той же папке, в которой лежит файл Проекта. Впрочем, пока ваш файл меньше 8 Мбайт или пока в нем не встретилось ошибочных рисунков, временные файлы вообще не создаются. В нижнем окне Substitute path prefix: вы можете указать путь, который будет использовать Help Workshop вместо пути, указанного в файле проекта. Этим можно воспользоваться, если вы, например, перенесли ваши файлы **.rtf** и **.bmp** на другой диск или на другой сервер. Тогда вместо исправления путей во всем тексте можно воспользоваться этой опцией.

Страница FTS позволяет создать файл **.fts**, для поиска по всему тексту. Впрочем, при желании пользователь может создать этот файл и сам, работая со справкой и выбрав страницу Поиск. Учтите, что этот файл может быть большого объема. Так что решайте, стоит ли создавать его.

Рис. 8.12

Задание имени скомпилированного файла справки



На странице Font вы можете выбрать шрифт, который будет использоваться в диалоговых окнах WinHelp, и осуществить замену шрифтов, использованных в файлах тем, на другие шрифты. Страница Macros дает возможность связать какие-то выделенные слова в тексте с соответствующим макросом. Страница Build Tags позволяет строить на основе одних и тех же файлов тем различные файлы справок с помощью описанных ранее сносков *.

8.3.1.9 Создание простого файла Проекта

Выше были рассмотрены богатые возможности, предоставляемые Help Workshop при создании файла Проекта справки. У читателя может создаться впечатление, что все это очень сложно. Однако в действительности при создании достаточно простых справок или на первых этапах создания даже сложных справок не требуется задания всех рассмотренных опций. Для создания файла Проекта простой справки достаточно выполнить следующие действия:

1. В меню File программы Help Workshop выполнить команду New и в открывшемся окне выбрать опцию Help Project. Далее в окне Project File Name надо задать имя и каталог файла Проекта справки (каталог выбрать тот, в котором лежит файл текстов тем .rtf).
2. В открывшемся окне Проекта (рис. 8.4) нажать кнопку Files и в окне Topic Files (рис. 8.5) щелкнуть на кнопке Add и выбрать среди файлов подготовленный файл текстов справки. После этого щелкнуть на ОК.

И это все. Вы можете компилировать ваш файл справки, как рассказано в разделе 8.3.2, и начать работать с ним. Все остальное — дело совершенствования этой справки, ее эстетического оформления и т.п.

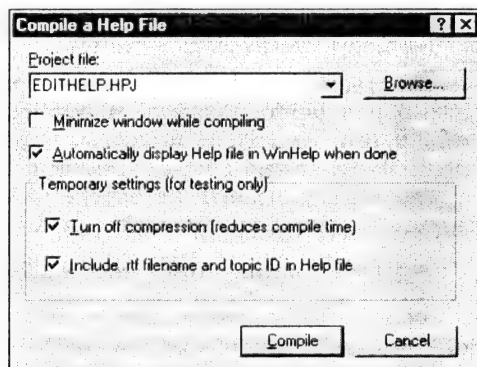
8.3.2 Компиляция и отладка справки

Когда файл Проекта создан, надо щелкнуть на кнопке Save and Compile в нижнем правом углу окна рис. 8.4. Файл проекта будет сохранен и откомпилирован, в результате чего создастся файл справки .hlp с тем же именем, что и файл проекта, или с именем, указанным вами на странице Files окна Options. Кнопкой Save and Compile надо пользоваться каждый раз, когда изменяется файл проекта. Если же в дальнейшем вы меняете только тексты в файле тем, то компиляцию удобнее про-

изводить командой File | Compile или проще — нажав соответствующую быструю кнопку с пиктограммой мясорубки (вторая справа на рис. 8.4). Причем вы можете это делать, даже не открывая предварительно файла Проекта. В результате выполнения этой команды откроется окно Compile a Help File (см. рис. 8.13), в котором вы можете выбрать нужный файл проекта и затем щелкнуть на кнопке Compile. Предварительно удобно установить флаг опции Automatically display Help file in WinHelp when done, которая обеспечивает сразу после компиляции просмотр скомпилированного файла справки. Если вы не установили опцию Minimize window while compiling, то во время компиляции вы будете видеть окно, с работающей мясорубкой, отражающей процесс компиляции. Установка указанной опции минимизирует это окно. Опция Turn off compression (reduces compile time) позволяет временно отключить сжатие файла. Это приведет к увеличению размера файла справки, но заметно уменьшит время компиляции больших справок. Опция Include .rtf filename and topic ID in Help file включит в файл справки имена файлов текстов тем и идентификаторы тем. Это позволит вам в процессе отладки, установив в меню File опцию Help Author, получать при просмотре справки оперативную информацию о каждом кадре справки.

Рис. 8.13

Задание опций компиляции файла справки

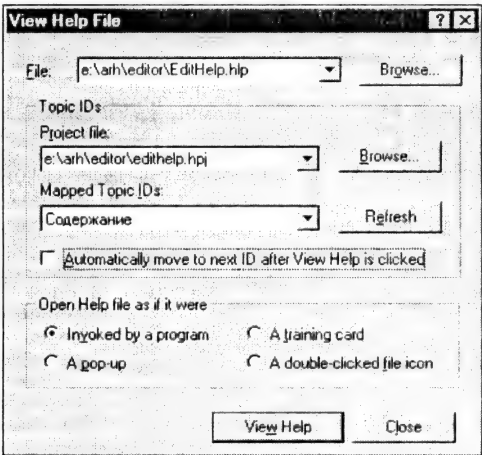


После компиляции вы увидите в окне Help Workshop содержимое текстового файла Compilation, в котором содержатся сведения о результатах компиляции. Здесь же будут замечания и сообщения об ошибках, которые возникнут при неправильном исходном файле .rtf. Однако, не спешите огорчаться, получив множество сообщений об ошибках. И не бросайтесь сразу искать их и исправлять в вашем файле тем. Чаще всего, несмотря на замечания и ошибки, файл справки все-таки компилируется и вы можете его посмотреть. В процессе этого просмотра и отладки вы скорее найдете ошибки, чем просто отыскивая их в исходном тексте.

После компиляции при включенной указанной выше опции Automatically display Help file in WinHelp when done вы сразу же можете поработать с откомпилированным файлом справки. В дальнейшем просмотр и работа с этим файлом может осуществляться или с помощью команды File | Run WinHelp, или соответствующей быстрой кнопкой со знаком вопроса (крайняя правая в панели инструментов на рис. 8.4). В результате откроется диалоговое окно View Help File (рис. 8.14), из которого запуск просмотра осуществляется кнопкой View Help. В верхнем окне File устанавливается интересующий файл справки. В окне Project file устанавливается имя файла проекта. Опция Mapped Topic IDs: позволяет выбрать ту из тем, указанных в разделе [MAP] файла Проекта, которая будет показываться первой. Таким образом вы можете моделировать обращение к справке из вашего приложения. Если вы к тому же установите опцию Automatically move to next ID after View Help is clicked и для начала зададите значение Mapped Topic IDs: равным первому из разделов [MAP], то при последовательных нажатиях на кнопку View Help вы поочередно пройдете весь список

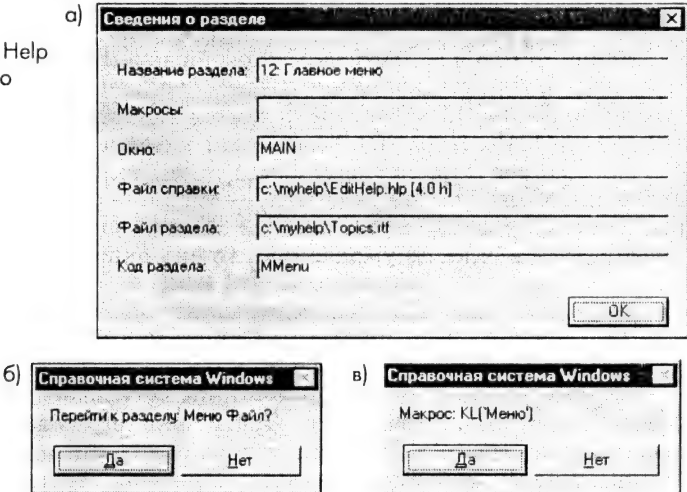
внешних входов справки. Опции в нижней части окна позволяют посмотреть, как выглядит справка при открытии ее разными способами. Опция *Invoked by a program* соответствует нормальному входу из программы. Опция *A pop-up* показывает, как выглядит тема при ее отображении во всплывающем окне. Опция *A training card* соответствует случаю, когда справка встроена в процесс отладки приложения по шагам (эта опция реализована не во всех версиях). Опция *A double-clicked file icon* соответствует входу в справку при двойном щелчке на ее пиктограмме.

Рис. 8.14
Задание опций просмотра файла справки



В процессе отладки удобно установить в меню *File* опцию *Help Author*. Тогда при просмотре справки вы можете в любом кадре нажать правую кнопку мыши, во всплывающем меню выбрать команду *Сведения о разделе* и увидеть окно (рис. 8.15 а), в котором указано название раздела (темы), окно, файл справки, исходный файл тестов (файл раздела) и идентификатор этого раздела. Сведения о файле и идентификаторе темы содержатся в этом окне, если вы при компиляции установили опцию *Include .rtf filename and topic ID in Help file*. В том же меню, всплывающем при нажатии правой кнопки мыши в режиме *Help Author*, вы можете установить опцию *Запросы при ссылках*. Тогда при каждом переходе от темы к теме будут появляться окна с запросами, вид которых представлен на рис. 8.15 б и в. Это позволит вам отследить, в каких случаях и к каким темам осуществляется переход.

Рис. 8.15
Окна отладки справки в режиме *Help Author*: сведения о разделе (а) и о переходах (б) и (в)

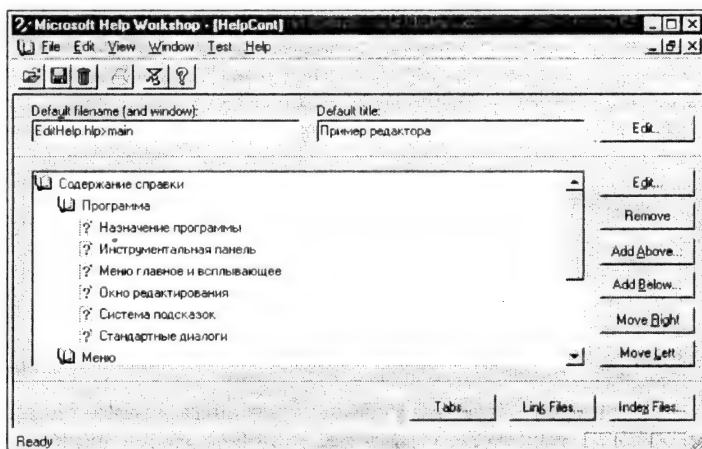


8.3.3 Файл содержания — .cnt

Файл Содержания (Contents file), имеющий расширение **.cnt**, является текстовым файлом, который можно проектировать с помощью Microsoft Help Workshop и присоединять его к файлу Проекта справки. Этот файл обеспечивает при работе со справкой страницу Содержание в окне справочной системы, в которой в виде пиктограмм открытых и закрытых книг и пиктограмм тем отражается структура справки. Чтобы создать новый файл содержания, надо в Help Workshop выполнить команду **File | New** и в окне **New** выбрать опцию **Help Contents**. Откроется окно с загруженным в него файлом содержания. Вид этого окна можно видеть на рис. 8.16; только в первый момент все его окна редактирования будут пустыми.

Рис. 8.16

Окно редактирования
файла Содержание



В верхних окнах надо задать: в левом — имя файла **.hlp** и окно по умолчанию (если оно определено в файле **.hlpj**), отделив имя окна символом **>**, в правом — заголовок, который будет появляться в окне Содержание при работе справочной системы. Заносить информацию в эти окна можно непосредственно, но проще щелкнуть на верхней кнопке **Edit** и работать в открывающемся при этом диалоговом окне.

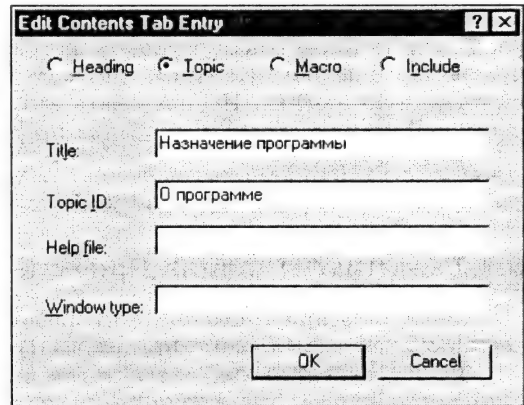
Нижнее окно заполняется с помощью кнопок **Add Above** и **Add Below**. Первая из этих кнопок вставляет новую строку выше той, в которой находится в данный момент курсор, а вторая — ниже этой строки. При нажатии любой из этих кнопок открывается окно **Edit Contents Tab Entry** (см. рис. 8.17), в котором вы можете задать очередной заголовок (**Heading**), отображающийся в виде книги, или очередную тему (**Topic**). Для заголовка указывается только его текст (**Tile**). Для темы записывается ее название (**Tile**), которое появится на странице Содержание справки, и идентификатор темы (**Topic ID**), указанный в файле тем. Если справка использует несколько файлов **.hlp**, то указывается также имя файла (**Help file**), содержащего данную тему. Может также указываться тип окна (**Window type**) — одного из тех, которые указаны в файле проекта (если они там указаны). Если отображение должно проводиться в окне по умолчанию, то вид окна не указывается.

В этом окне можно также задать макрос (**Macro**) или включить внешний файл содержания (**Include**). Последнее позволяет легко видоизменять при необходимости файл содержания, не переписывая его целиком, а включая в него какие-то дополнения, записанные в отдельных файлах.

Вернемся к рис. 8.16 и рассмотрим остальные кнопки окна редактирования файла Содержания. Кнопки **Move Right** и **Move Left** позволяют сдвигать вправо и влево выделенные строки заголовков и тем, обеспечивая нужные отступы, характеризующие многоуровневую структуру. При этом строки тем, относящихся к одному

Рис. 8.17

Задание очередного элемента файла
Содержание



заголовок, располагаются на один шаг правее своего заголовка и не могут сдвигаться друг относительно друга. Точнее, сдвинуть их можно, но это никак не отразится на их расположении при работе пользователя со справкой.

Кнопка Edit позволяет редактировать выделенную строку, а кнопка Remove удаляет строку.

В Help Workshop отсутствует возможность перемещать строки вверх или вниз, упорядочивая их последовательность. Подобные перемещения можно делать, открыв файл Содержания как текстовый в Word и переместив нужные строки средствами текстового редактора.

8.4 Особенности создания справки, работающей на любых версиях Windows

8.4.1 Ограничения возможностей справки

Выше была рассмотрена программа Help Workshop, позволяющая быстро и просто разрабатывать и отлаживать файлы Проекта и Содержания справки. При работе с этой программой можно особо не задумываться над синтаксисом файлов. Но разработанные с помощью этого инструмента справки годятся только для 32-разрядных Windows. Они не смогут быть прочитаны средствами Windows 3.x. Поэтому, если вы делаете приложение, рассчитывая на его использование, в частности, и в более ранних версиях Windows, то и справку к этому приложению надо делать средствами Windows 3.x.

Конечно, ориентация на любые, в том числе ранние версии Windows приводит к определенным ограничениям возможностей справки. Основные из этих ограничений следующие:

- Нельзя отображать содержание в виде дерева с изображениями закрытых и открытых книг, как это делается в 32-разрядных Windows. Соответственно не надо писать файл Содержания .cnt.
- В файле тем нельзя использовать сноски А.
- Сноски К в файле тем имеют только один уровень. Соответственно каждая такая сноска может содержать только одно слово или словосочетание.
- Нет многих макросов, в частности, макросов KLink и ALink. Вместо них используется макрос

```
JumpKeyword("<имя файла .rtf>", "<ключевое слово К-сноски>").
```

Файл Проекта приходится составлять вручную с помощью текстового редактора. Компиляция справки осуществляется программами HC31 или HCP. Последняя программа предпочтительнее, так как позволяет использовать расширенную и виртуальную память. Использовать эти программы очень просто: надо передать через командную строку имя файла проекта **.hpf**. Например, команда:

```
HC31 MYHELP.HPJ
```

создаст файл справки MYHELP.HLP.

8.4.2 Синтаксис файла Проекта

Поскольку при ориентации на любые версии Windows вам придется вручную составить текстовый файл **.hpf**, рассмотрим коротко синтаксис этого файла.

Файл создается в любом текстовом редакторе и сохраняется в виде текста. Расширение файла — **.hpf**, а имя файла должно совпадать с именем результирующего файла справки **.hlp**. При использовании редактора Word для сохранения этого файла надо выполнить команду Файл | Сохранить как и в открывшемся диалоговом окне установить опцию Тип документа, равной Только текст, а имя файла не забыть указать с расширением **.hpf**, так как иначе программа подставит расширение по умолчанию — **.txt**.

В текст файла можно вставлять комментарии. Признаком строки комментария является предшествующий ей символ точки с запятой (;).

Файл состоит из ряда разделов, заголовки которых заключаются в квадратные скобки. Головной раздел [OPTIONS] может включать в себя следующие опции (все они не обязательны и могут отсутствовать).

Опция

```
CONTENTS=<идентификатор кадра содержания>
```

задает идентификатор темы, которая обычно включает в себя перечень разделов справки. Это та тема, которая открывается при запуске справки и на которую попадает пользователь, работающий со справкой, при нажатии кнопки Содержание. По умолчанию (если эта опция не задана) темой, определяющей содержание, является первая тема первого файла тем **.rtf**.

Опция

```
TITLE=<заголовок>
```

определяет строку, появляющуюся в заголовке окна справки. Если вы ее не задали, то будет использоваться стандартный заголовок.

Опция

```
COMPRESS=<уровень сжатия>
```

задает уровень сжатия файла справки при его компиляции. Чем выше уровень сжатия, тем меньше размер результирующего файла справки, но больше время компиляции. Поэтому в процессе отладки больших файлов справки целесообразно использовать низкий уровень сжатия, а отлаженную справку откомпилировать с высоким уровнем сжатия. Небольшие справки можно компилировать сразу с высоким уровнем сжатия. В опции COMPRESS можно задавать следующие значения уровней:

FALSE , или 0 , или NO	Быстрая компиляция без сжатия. Большой объем результирующего файла
MEDIUM	Средний уровень скорости компиляции и размера результирующего файла
HIGH , или TRUE , или 1 , или YES	Медленная компиляция с высокой степенью сжатия. Минимальный размер результирующего файла

По умолчанию компиляция производится без сжатия.

Опция

ERRORLOG=<файл протокола>

задает имя файла протокола (принятое расширение — .log), в котором записываются ошибки, обнаруженные при компиляции. Этот файл нужен только в процессе отладки. Если опция ERRORLOG не задана, то сообщения об ошибках отображаются на экране, но в файл не заносятся. Если ошибок много, то они не поместятся на экране и отлаживать будет очень сложно. К тому же файл протокола удобно открыть в Word одновременно с файлами тем, что ускорит вам поиск и исправление ошибок. Поэтому можно рекомендовать при отладках больших справок задавать опцию ERRORLOG.

Из сказанного следует, что раздел [OPTIONS] в простейшем случае может вообще не задаваться, поскольку он не содержит ни одной обязательной опции.

Не обязательный раздел [CONFIG] может использоваться для указания макросов, которые должны выполняться при открытии файла справки. Например, вы можете указать макросы, включающие какие-то разделы меню или кнопки, отсутствующие в стандартном окне справочной системы:

```
[CONFIG]
; Включается для кнопки "История"
CreateButton("History","&История", "History()")
BrowseButtons() ; Включается для появления кнопок "<<", ">>"
InsertMenu("mexit"," Выход",5) ; Добавляет меню "Выход"
AppendItem("mexit","exit","Выход","exit()"); Раздел "Выход"
```

Раздел [FILES] включает в себя перечень файлов тем .rtf. Это единственный раздел, который обязательно должен присутствовать в файле Проекта справки. Имя каждого файла .rtf начинается с новой строки. Для простой справки с одним файлом тем это может быть всего один файл.

Раздел [MAP] содержит таблицу соответствия идентификаторов тем, используемых при написании файла .rtf, и номеров контекстной справки, используемых в вашем приложении для ссылок на эти темы. Например, если в файле .rtf имеется тема с идентификатором menu, на которую из приложения имеется контекстная ссылка 2, и тема с идентификатором Содержание, на которую в приложении имеется ссылка по номеру 1, то раздел [MAP] может иметь вид:

```
[MAP]
menu          2 ; обращение к справке по меню
Содержание    1 ; обращение к содержанию
```

После точек с запятой записаны комментарии, упрощающие понимание таблицы соответствия идентификаторов и номеров.

Раздел [WINDOWS] описывает окна справки — основное (main) и вторичные (если они используются). Описание каждого окна имеет вид:

```
<тип> = "<заголовок>", (<x>,<y>,<ширина>,<высота>),
        <развертывание>, (<цвет окна>),
        (<цвет не прокручиваемой части>)
```

Первый элемент этого описания — заголовок окна. Для главного окна он не указывается, поскольку задается описанной выше опцией TITLE в разделе [OPTIONS]. Для вторичных окон заголовок тоже может не задаваться. Следующий элемент описания — координаты левого верхнего угла окна и его размеры. Следующий элемент — <развертывание>, может принимать значения 0 (не развертываемое на весь экран) или 1 (развертываемое). Далее следуют два элемента описания, определяющие цвета прокручиваемой и не прокручиваемой частей окна. Каждый цвет задается тремя целыми числами, описывающими интенсивность красного, зеленого и синего. Максимальная интенсивность — 255, минимальная — 0. Например, (255, 255, 255) — белый цвет, (127, 127, 127) — темно серый. Приведем пример раздела [WINDOWS]:


```
[WINDOWS]
main=(150,50,800,800),0,(255,255,255),(127,127,127)
W1="Пояснения", (10,10,400,400),0,(255,255,255),(255,255,255)
```

Раздел [WINDOWS] может отсутствовать, если справка не использует вторичных окно. Тогда атрибуты главного окна берутся по умолчанию.

Кроме перечисленных, в файле Проекта могут быть еще разделы [ALIAS], [BUILDTAGS], [BITMAPS], [BAGGAGE]. Эти разделы используются реже, чем описанные выше, и на них мы не будем останавливаться.

В заключение приведем примеры файлов Проекта. Как было видно из изложенного выше, в простейшем случае файл может состоять всего из одного раздела, содержащего имя одного файла .rtf:

```
[FILES]
MyTopic.rtf
```

Ниже приведен пример более развернутого файла:

; Файл проекта справки MyHelp.

```
[OPTIONS]
ERRORLOG='Errors' ;Файл протокола с сообщениями об ошибках.
CONTENTS= Содержание;Головная тема.Включать не обязательно.
TITLE='Тест справки';Заголовок главного окна.
COMPRESS=High
```

```
[FILES]
Topics1.rtf ;Файл тем.
```

```
[MAP]
menu 1 ;Обращение к справке по меню
Содержание 2 ;Обращение к содержанию
```

```
[WINDOWS]
main=(150,50,800,800),0,(255,255,255),(127,127,127)
W1="Пояснения", (10,10,400,400),0,(255,255,255),(255,255,255)
```

```
[CONFIG]
;Кнопка "История"
CreateButton("History","&История","History()")

;Кнопки "<<",">>"
BrowseButtons()

;Меню "Выход"
InsertMenu("mexit","Выход",5)

;Раздел "Выход"
AppendItem("mexit","exit","Выход","exit()")

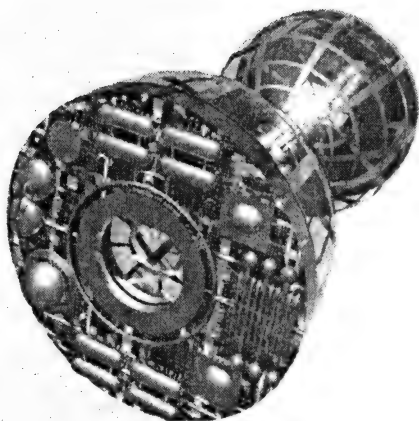
;Раздел "Как использовать справку"
InsertItem("mnu_help","How","Как использовать справку","HelpOn()",0)
```

В заключение следует отметить, что, хотя при разработке справок, пригодных для любых версий Windows, нельзя использовать программу Help Workshop для отладки справки, может оказаться полезным все-таки применять ее для предварительного составления файла Проекта. При этом облегчается, например, задание атрибутов окон и многие другие операции. Однако после того, как файл сформирован и справка отлажена, следует открыть в любом текстовом редакторе файл Проекта и убрать из него некоторые операторы, которые не смогут понять программы компиляции НС31 и НСР. Затем надо скомпилировать файл справки с помощью этих программ. Какие именно операторы следует убрать, вы сможете увидеть по тем ошибкам, которые будут отмечены в процессе компиляции.

Часть III

Создание приложений для работы с базами данных

- Глава 9 Приложения для работы с локальными базами данных
- Глава 10 Создание приложений для работы с базами данных в сети
- Глава 11 Обработка и документирование данных



Приложения для работы с локальными базами данных

9.1 Базы данных

9.1.1 Принципы построения баз данных

Всегда, когда возникает потребность манипулировать большими массивами данных, используются базы данных. Работа с базами данных в C++Builder — это столь обширная тема, что ей надо было бы посвящать отдельную книгу объемом не меньшую, чем та, которую вы сейчас читаете. Поэтому в рамках данной книги мы вынуждены будем рассмотреть только основы создания приложений для работы с базами данных. Впрочем, этого будет достаточно для того, чтобы строить достаточно мощные и полезные приложения.

Допуская, что читатели неплохо знакомы с принципами построения баз данных, мы тем не менее очень коротко рассмотрим здесь эти принципы, чтобы в дальнейшем использовать единую и понятную всем терминологию.

База данных — (мы будем говорить о так называемых *реляционных базах данных*) это прежде всего набор таблиц, хотя, как мы увидим позднее, в базу данных могут входить также процедуры и ряд других объектов. *Таблицу* можно представить себе как обычную двумерную таблицу с характеристиками (атрибутами) какого-то множества объектов. Таблица имеет *имя* — идентификатор, по которому на нее можно сослаться. В табл. 9.1 приведен пример фрагмента подобной таблицы с именем Pers, содержащей сведения о сотрудниках некоторой организации. Эта таблица будет в дальнейшем использоваться в примерах по работе с базами данных. Вы можете найти ее на диске, приложенном к книге, или построить сами при изучении материала раздела 9.2.

Таблица 9.1. Пример таблицы данных о сотрудниках Pers

Номер	Отдел	Фамилия	Имя	Отчество	Год рождения	Пол	Характеристика	Фотография
Num	Dep	Fam	Nam	Pat	Year_b	Sex	Charact	Photo
1	Бухгалтерия	Иванов	Иван	Иванович	1950	м
2	Цех 1	Петров	Петр	Петрович	1960	м
3	Цех 2	Сидоров	Сидор	Сидорович	1955	м
4	Цех 1	Иванова	Ирина	Ивановна	1961	ж
...

Столбцы таблицы соответствуют тем или иным характеристикам объектов — *полям*. Каждое поле характеризуется именем и типом хранящихся данных. *Имя поля* — это идентификатор, который используется в различных программах для манипуляции данными. Он строится по тем же правилам, как любой идентифика-

тор, т.е. пишется латинскими буквами, состоит из одного слова и т.д. Таким образом имя — это не то, что отображается на экране или в отчете в заголовке столбца (это отображение естественно писать по-русски), а идентификатор, соответствующий этому заголовку. Например, для таблицы 9.1 введем для последующих ссылок имена полей **Num**, **Dep**, **Fam**, **Nam**, **Par**, **Year_b**, **Sex**, **Charact**, **Photo**, соответствующие указанным в ней заголовкам полей.

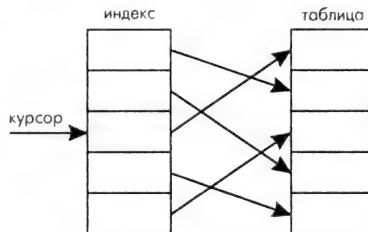
Тип поля характеризует тип хранящихся в поле данных. Это могут быть строки, числа, булевы значения, большие тексты (например, характеристики сотрудников), изображения (фотографии сотрудников) и т.п.

Каждая *строка таблицы* соответствует одному из объектов. Она называется *записью* и содержит значения всех полей, характеризующие данный объект.

При построении таблиц баз данных важно обеспечивать непротиворечивость информации. Обычно это делается введением *ключевых полей* — обеспечивающих уникальность каждой записи. Ключевым может быть одно или несколько полей. В приведенном выше примере можно было бы сделать ключевыми совокупность полей **Fam**, **Nam** и **Par**. Но в этом случае нельзя было бы заносить в таблицу сведения о полных однофамильцах, у которых совпадают фамилия, имя и отчество. Поэтому в таблицу введено первое поле **Num** — номер, которое можно сделать ключевым, обеспечивающим уникальность каждой записи.

Рис. 9.1.

Схема перемещения курсора по индексу



При работе с таблицей пользователь или программа как бы скользит курсором по записям. В каждый момент времени есть некоторая *текущая запись*, с которой и ведется работа. Записи в таблице базы данных физически могут располагаться без какого-либо порядка, просто в последовательности их ввода (появления новых сотрудников). Но когда данные таблицы предъявляются пользователю, они должны быть упорядочены. Пользователь может хотеть просматривать их в алфавитном порядке, или рассортированными по отделам, или по мере нарастания года рождения и т.п. Для упорядочивания данных используется понятие *индекса*. Индекс показывает, в какой последовательности желательно просматривать таблицу. Он является как бы посредником между пользователем и таблицей (см. рис. 9.1).

Курсор скользит по индексу, а индекс указывает на ту или иную запись таблицы. Для пользователя таблица выглядит упорядоченной, причем он может сменить индекс и последовательность просматриваемых записей изменится. Но в действительности это не связано с какой-то перестройкой самой таблицы и с физическим перемещением в ней записей. Меняется только индекс, т.е. последовательность ссылок на записи.

Индексы могут быть *первичными* и *вторичными*. Например, первичным индексом могут служить поля, отмеченные при создании базы данных как ключевые. А вторичные индексы могут создаваться из других полей как в процессе создания самой базы данных, так и позднее в процессе работы с ней. Вторичным индексам присваиваются имена — идентификаторы, по которым их можно использовать.

Если индекс включает в себя несколько полей, то упорядочивание базы данных сначала осуществляется по первому полю, а для записей, имеющих одинаковые значения первого поля — по второму и т.д. Например, базу данных персонала можно индексировать по отделам, а внутри каждого отдела — по алфавиту.

База данных обычно содержит не одну, а множество таблиц. Например, база данных о некоторой организации может содержать таблицу имеющихся в ней подразделений с характеристикой каждого из них. Пример такой таблицы с именем **Dep**, которая будет использоваться нами в дальнейшем, приведен в таблице 9.2. Имена полей этой таблицы, которые в дальнейшем мы будем использовать: **Dep** и **Proisv**.

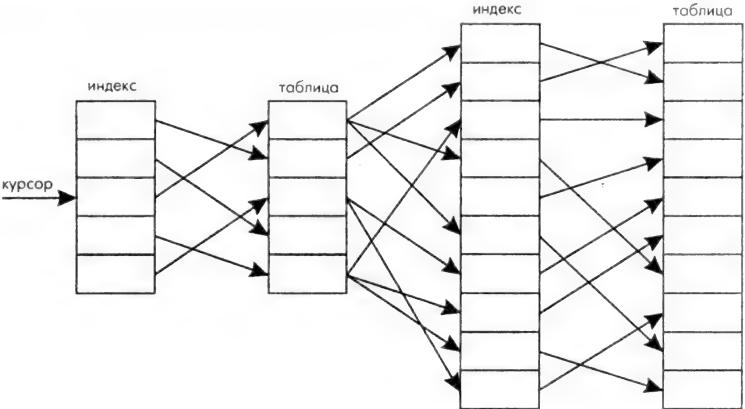
Таблица 9.2. Пример таблицы данных о подразделениях **Dep**

Отдел	Тип
Dep	Proisv
Бухгалтерия	управление
Цех 1	производство
Цех 2	производство

Отдельные таблицы, конечно, полезны, но гораздо больше информации можно извлечь именно из совокупности таблиц. Например, пользователю может потребоваться узнать общее количество сотрудников, работающих в производственных цехах. Но ни одна из приведенных выше таблиц не поможет ответить на этот вопрос, поскольку в таблице **Pers** отсутствуют сведения о типах отделов, а в таблице **Dep** — о сотрудниках. Для получения ответов на подобные запросы необходимо рассмотрение совокупности связанных таблиц.

В связанных таблицах обычно одна выступает как *главная*, а другая или несколько других — как вспомогательные, управляемые главной. В этом случае взаимодействие таблиц иллюстрируется рисунком 9.2. Главная и вспомогательная таблицы связываются друг с другом *ключом*. В качестве ключа могут выступать какое-то поля, присутствующие в обеих таблицах. Например, в приведенных ранее таблицах головной может быть таблица **Dep**, вспомогательной **Pers**, а связываться они могут по полю **Dep**, присутствующему в обеих таблицах. Курсор скользит по индексу главной таблицы. Каждой записи в главной таблице ключ ставит в соответствие в общем случае множество записей вспомогательной таблицы. Так в нашем примере каждой записи главной таблицы **Dep** соответствуют те записи вспомогательной таблицы **Pers**, в которых ключевое поле **Dep** с названием отдела совпадает с названием отдела в текущей записи главной таблицы. Иначе говоря, если в текущей записи главной таблицы в поле **Dep** написано «Бухгалтерия», то во вспомогательной таблице **Pers** выделяются все записи сотрудников бухгалтерии.

Рис. 9.2.
Схема взаимодействия
главной и
вспомогательной
таблицы



Создают базы данных и обрабатывают запросы к ним *системы управления базами данных* — СУБД. Известно множество СУБД, различающихся своими возможностями или обладающих примерно равными возможностями и конкурирующих друг с другом: Paradox, dBase, Microsoft Access, FoxPro, Oracle, InterBase, Sybase и много других.

Разные СУБД по-разному организуют и хранят базы данных. Например, Paradox и dBase используют для каждой таблицы отдельный файл. В этом случае база данных — это каталог, в котором хранятся файлы таблиц. В Microsoft Access и в InterBase несколько таблиц хранится как один файл. В этом случае база данных — это имя файла с путем доступа к нему. Системы типа клиент/сервер, такие, как серверы Sybase или Microsoft SQL, хранят все данные на отдельном компьютере и общаются с клиентом посредством специального языка, называемого SQL (см. раздел 10.1).

Поскольку конкретные свойства баз данных очень разнообразны, пользователю было бы весьма затруднительно работать, если бы он должен был указывать в своем приложении все эти каталоги, файлы, серверы и т.п. Да и приложение часто пришлось бы переделывать при смене, например, структуры каталогов и при переходе с одного компьютера на другой. Чтобы решить эту проблему, используют псевдонимы баз данных. *Псевдоним (alias)* содержит всю информацию, необходимую для обеспечения доступа к базе данных. Эта информация сообщается только один раз при создании псевдонима. А приложение для связи с базой данных использует псевдоним. В этом случае приложению безразлично, где физически расположена та или иная база данных, а часто безразлична и СУБД, создавшая и обслуживающая эту базу данных. При смене системы каталогов, сервера и т.п. ничего в приложении переделывать не надо. Достаточно, чтобы администратор базы данных ввел соответствующую информацию в псевдоним.

При работе с базами данных часто используется *кэширование* всех изменений. Это означает, что все изменения данных, вставка новых записей, удаление существующих записей, т.е. все манипуляции с данными, проводимые пользователем, сначала делаются не в самой базе данных, а запоминаются в памяти во временной, виртуальной таблице. И только по особой команде после всех проверок правильности вносимых в таблицу данных пользователю предоставляется возможность или зафиксировать все эти изменения в базе данных, или откатиться от этого и вернуться к тому состоянию, которое было до начала редактирования.

Фиксация изменений в базе данных осуществляется с помощью *транзакций*. Это совокупность команд, изменяющих базу данных. На протяжении транзакции пользователь может что-то изменять в данных, но это только видимость. В действительности все изменения сохраняются в памяти. И пользователю предоставляется возможность завершить транзакцию или внесением всех изменения в реальную базу данных, или отказом от этого с возвратом к тому состоянию, которое было до начала транзакции.

9.1.2 Типы баз данных

Для разных задач целесообразно использовать различные модели баз данных, поскольку, конечно, базу данных сведений о сотрудниках какого-то небольшого коллектива и базу данных о каком-нибудь банке, имеющем филиалы во всех концах страны, надо строить по-разному.

Процесс определения того, какая база данных более подходит для конкретного приложения, называется *масштабированием*. Это сложная задача, которую мы не будем затрагивать. Однако, прежде, чем двигаться дальше, необходимо иметь представление о возможных моделях баз данных, поскольку это влияет на построение приложений в C++Builder.

В следующих разделах коротко рассмотрены четыре модели баз данных:

- Автономные
- Файл-серверные
- Клиент/сервер
- Многоярусные

Прежде, чем переходить к рассмотрению различных моделей баз данных, отметим, что работа с данными в C++Builder в основном осуществляется через Borland Database Engine (BDE) — процессор баз данных фирмы Borland. Соответствующая программа должна быть поставлена на компьютере пользователя во всех моделях баз данных, кроме многоярусных.

9.1.2.1 Автономные базы данных

Автономные локальные базы данных являются наиболее простыми. Они хранят свои данные в локальной файловой системе на том компьютере, на котором установлены; система управления и машина базы данных, осуществляющая к ним доступ, находится на том же самом компьютере. Сеть не используется. Поэтому разработчику автономной базы данных не приходится иметь дело с проблемой параллельного доступа, когда два человека пытаются одновременно изменить одну и ту же запись, потому что такого никогда не может быть. Вообще, автономные базы данных не используются для приложений, требующих значительной вычислительной мощности, потому что процессорное время будет потрачено на выполнение манипуляций с данными и в целом будет потеряно для приложения.

Автономные базы данных полезны для развития тех приложений, которые распространены среди многих пользователей, каждый из которых поддерживает отдельную базу данных. Это, например, приложения, обрабатывающие документацию небольшого офиса, кадровый состав небольшого предприятия, бухгалтерские документы небольшой бухгалтерии. Каждый пользователь такого приложения манипулирует своими собственными данными на своем компьютере. Пользователю нет необходимости иметь доступ к данным любого другого пользователя, так что отдельная база данных здесь вполне приемлема.

9.1.2.2 Файл-серверные базы данных

Файл-серверные базы данных отличаются от автономных тем, что они могут быть доступны многим клиентам через сеть. Это очень удобно, так как изменения в таких базах данных видят все пользователи. Например, базу данных сотрудников крупного учреждения целесообразно делать именно такой, чтобы администраторы отдельных подразделений обращались к ней, а не заходили у себя локальные базы данных (при этом можно сделать так, чтобы каждый администратор видел только ту информацию, которая относится к его подразделению).

Сама база данных хранится на сетевом файл-сервере в единственном экземпляре. Для каждого клиента во время работы создается локальная копия данных, с которой он манипулирует. При этом возникают (и решаются) проблемы, связанные с возможным одновременным доступом нескольких пользователей к одной и той же информации. Например, при проектировании приложений, работающих с подобными базами данных, должны быть решены такие проблемы: что делать, если пользователь прочел некоторую запись и, пока он ее просматривает и собирается изменить, другой пользователь меняет или удаляет эту запись.

Одним из недостатков баз данных файл-сервер является непроизводительная загрузка сети. При каждом запросе клиента данные в его локальной копии полностью обновляются из базы данных на сервере. Даже если запрос относится всего к одной записи, обновляются все записи данных. Если записей в базе данных много, то даже при небольшом числе клиентов сеть будет загружена очень основательно, что серьезно скажется на скорости выполнения запросов.

Другой недостаток связан с тем, что забота о целостности данных при такой организации работы возлагается на программы клиентов. Если они недостаточно тщательно продуманы, в базу данных легко занести ошибки, которые могут отразиться на всех пользователях.

9.1.2.3 Базы данных клиент/сервер

Для больших баз данных с множеством пользователей часто используются базы данных на платформе клиент/сервер. В этом случае доступ к базе данных для группы клиентов выполняется специальным компьютером — сервером. Клиент дает задание серверу выполнить те или иные операции поиска или обновления базы данных. И мощный сервер, ориентированный на операции с запросами самым оптимальным способом, выполняет их и сообщает клиенту результаты своей работы.

Подобная организация работы повышает эффективность выполнения приложений за счет использования мощности сервера, разгружает сеть, обеспечивает хороший контроль целостности данных.

В базах данных клиент/сервер возникает дополнительная проблема — спроектировать приложение так, чтобы оно максимально использовало возможности сервера и минимально нагружало сеть, передавая через нее только минимум информации.

9.1.2.4 Многоярусные базы данных

Это новый и многообещающий путь обработки данных в сети. Иногда (в частности, в документации C++Builder) этот способ организации баз данных называется multitier — многонитевые. В этом термине под нитью понимается один из множества потоков данных, обменивающихся одновременно с базой данных.

Наиболее распространен трехярусный вариант:

- На нижнем уровне на компьютерах пользователя расположены приложения клиентов, обеспечивающие пользовательский интерфейс.
- На втором уровне расположен сервер приложений, обеспечивающий обмен данными между пользователями и распределенными базами данных. Сервер приложений размещается в узле сети, доступном всем клиентам
- На третьем уровне расположен удаленный сервер баз данных, принимающий информацию от серверов приложений и управляющий ими.

Подобную концепцию обработки данных пропагандируют, в частности, фирмы Oracle и Sun. Первый, элементарный уровень состоит из «тонких клиентов», то есть несложных терминалов, предназначенных, в основном, для ввода — вывода. Второй, средний (middleware) уровень — это рабочие станции и серверы приложений, то есть значительно более серьезные машины, на которых выполняются программы, критичные к загрузке процессора. Третий и последний уровень — мощные специализированные серверы баз данных.

Отметим одну особенность многоярусных распределенных баз данных: в них на нижнем уровне — на компьютерах пользователя не требуется установка Borland Database Engine (BDE). В этом заключается одно из преимуществ такой организации баз данных.

9.1.3 Технологии COM и CORBA

Для создания распределенных СУБД в настоящее время используются в основном две технологии: COM и CORBA. Кроме того в связи с бурным развитием Интернет все большее значение приобретает технология Web, которая, возможно, в будущем станет определяющей.

В рамках данной книги распределенные СУБД рассматриваться не будут. Соответственно, не будут рассматриваться и связанные с построением таких баз данных компоненты **Provider**, **ClientDataSet**, **RemoteServer** и др. Тем не менее, ниже даются некоторые основные понятия, связанные с созданием распределенных приложений, поскольку без этого картина организации работы с базами данных была бы неполной.

Технология *COM* (*компонентная модель объекта*) разработана корпорацией Microsoft. Ее назначение — предоставление возможности одной программе (клиенту) работать с объектом другой программы (сервера). COM — это модель объекта, которая предусматривает полную совместимость во взаимодействии между компонентами, написанными разными компаниями и на разных языках. При этом клиент и сервер могут располагаться на разных компьютерах.

На COM основаны такие технологии Microsoft, как OLE, Automation OLE, ActiveX, OCX. На COM построен, например, весь интерфейс Windows 98.

COM определяет унифицированный двоичный *интерфейс*, полностью независимый от языка программирования, использованного при реализации компонента. Компонент, написанный в соответствии со спецификациями двоичного интерфейса COM, может вступать во взаимодействие с другим компонентом, не зная в действительности ничего о реализации последнего.

Каждый интерфейс имеет уникальный идентификатор IID (Interface Identifier), являющийся частным случаем GUID (Global Unique Identifier) — глобального идентификатора, используемого в Windows. Параметры интерфейса описывают некоторый класс с идентификатором CLSID (Class ID), в котором объявлены поля, свойства, методы, параметры обращения к свойствам и методам. Таким образом, клиент с помощью этого интерфейса может использовать объект COM сервера так же, как свой собственный объект. Любой объект COM имеет интерфейс IUnknown, с помощью которого получает доступ к основному интерфейсу объекта.

Сервер COM реализуется в виде программы или DLL. Если клиент и сервер располагаются на разных компьютерах, используется распределенный вариант COM — DCOM. При этом обмен информацией между клиентом и сервером осуществляется двумя промежуточными программами: Proxy (уполномоченный) и Stub (заглушка). Proxy располагается на машине клиента. Получив от клиента запрос, Proxy упаковывает его в пакет COM и переправляет на машину сервера. Там этот пакет перехватывает Stub, распаковывает пакет и передает запрос серверу. Таким образом, запрос выполняется на машине сервера.

CORBA (*Common Object Request Broker Architecture*) — это стандарт построения приложений с распределенными объектами. Разработчиком CORBA является отраслевой комитет OMG (Object Management Group — группа управления объектами), представляющий многие фирмы. Взаимодействие клиента и сервера в CORBA осуществляется через ряд посредников. На машине клиента размещается Stub и ORB (Object Request Broker). Клиент обращается к Stub так, как если бы это был сам объект. При этом используется интерфейс объекта, предоставляемый клиенту с помощью Stub.

Stub транслирует полученный от клиента вызов какого-то метода объекту ORB, а тот передает соответствующее сообщение в сеть. В сетевом окружении клиента располагаются объекты Smart Agent (интеллектуальные агенты). Сообщение, переданное объектом ORB, перехватывает один из Smart Agent, отыскивает сетевой адрес соответствующего сервера и передает полученное сообщение на машину сервера. На машине сервера сообщение воспринимает расположенный там объект ORB. Он передает его расположенному там же объекту BOA (Basic Object Adapter — базовый адаптер объекта). BOA может проводить фильтрацию запросов, определяя, какой уровень доступа разрешен данному клиенту и вообще разрешено ли обрабатывать такой запрос данного клиента. Если доступ разрешен, то BOA передает вызов в особый объект сервера — Skeleton, который и реализует сам вызов.

Интерфейс в CORBA описывается с помощью специального языка IDL (Interface Definition Language), напоминающего C++. Компилятор IDL создает в процессе компиляции объекты Stub и Skeleton данного интерфейса. Использование языка высокого уровня в сочетании с компилятором позволяет делать обмен данными независимым от аппаратных средств. Поэтому клиент и сервер могут располагаться на машинах разных платформ, например, на персональном компьютере IBM и рабочей станции Sun.

При обмене информацией между ORB и Smart Agent используется протокол UDP. Для реализации CORBA в сетевом окружении клиента должен существовать хотя бы один объект Smart Agent. Обычно в локальной сети Smart Agent располагается на головной машине, а в Интернет — на одном из узлов. При создании сервера он регистрируется в Smart Agent. Таким образом Smart Agent знает, где найти тот или иной сервер, а при отказе одного сервера может переключиться на другой. Это повышает надежность работы.

9.1.4 Организация связи с базами данных в C++Builder

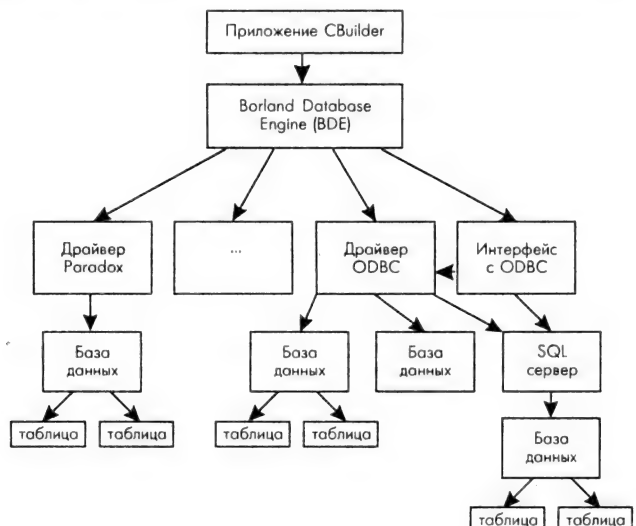
Основой работы C++Builder с базами данных является Borland Database Engine (BDE) — процессор баз данных фирмы Borland. BDE служит посредником между приложением и базами данных. Он предоставляет пользователю единый интерфейс для работы, развязывающий пользователя от конкретной реализации базы данных. Благодаря этому не надо менять приложение при смене реализации базы данных. Приложение C++Builder никогда не обращается непосредственно к базе данных, а только к BDE. Таким образом, общение с базами данных соответствует схеме, приведенной на рис. 9.3.

Приложение C++Builder, когда ему нужно связаться с базой данных, обращается к BDE и сообщает обычно псевдоним базы данных и необходимую таблицу в ней. BDE реализован в виде динамически присоединяемых библиотек DLL (файлы IDAPI01.DLL, IDAPI32.DLL). Они, как и любые библиотеки, снабжены API (Application Program Interface — интерфейсом прикладных программ), названным IDAPI (Integrated Database Application Program Interface). Это список процедур и функций для работы с базами данных, которым и пользуются приложения.

BDE по псевдониму находит подходящий для указанной базы данных драйвер. *Драйвер* — это вспомогательная программа, которая понимает, как общаться с базами данных определенного типа. Если в BDE имеется собственный драйвер со-

Рис. 9.3.

Схема связи приложения C++Builder с базами данных



ответствующей СУБД, то BDE связывается через него с базой данных и с нужной таблицей в ней, обрабатывает запрос пользователя и возвращает в приложение результаты обработки. BDE поддерживает естественный доступ к таким базам данных, как Microsoft Access, FoxPro, Paradox, dBase.

Если собственного драйвера нужной СУБД в BDE нет, то используется драйвер ODBC. ODBC (Open Database Connectivity) — это DLL, аналогичная по функциям BDE, но разработанная фирмой Microsoft. Она хранится в файле ODBC.DLL. Поскольку Microsoft включила поддержку ODBC в свои офисные продукты и для ODBC созданы драйверы практически к любым СУБД, фирма Borland включила в BDE драйвер, позволяющий использовать ODBC. Правда, работа через ODBC осуществляется несколько медленнее, чем через собственные драйверы СУБД, включенные в BDE. Но благодаря связи с ODBC масштабируемость C++Builder существенно увеличилась и сейчас из C++Builder можно работать с любой сколько-нибудь значительной СУБД.

BDE поддерживает SQL — стандартизованный язык запросов, позволяющий обмениваться данными с SQL-серверами, такими, как Sybase, Microsoft SQL, Oracle, Interbase. Эта возможность используется особенно широко при работе на платформе клиент/сервер и в распределенных базах данных.

В C++Builder 5 введена другая альтернативная возможность работы с базами данных, минуя BDE. Это разработанная в Microsoft технология ActiveX Data Objects (ADO). ADO — это пользовательский интерфейс к любым типам данных, включая реляционные и не реляционные базы данных, электронную почту, системные, текстовые и графические файлы. Связь с данными осуществляется посредством так называемой технологии OLE DB.

Использование ADO обеспечивает более эффективную работу с данными. Однако, надо сказать, что возможности ADO в C++Builder пока в некоторых отношениях ниже, чем возможности BDE. Поэтому в дальнейшем мы в основном сосредоточимся на работе с BDE. А особенности использования ADO будут рассмотрены в главе 10 в разделе 10.4.

Еще один альтернативный доступ к базам данных Interbase введен в C++Builder 5 на основе технологии InterBase Express (IBX). В библиотеке компонентов C++Builder 5 имеется страница InterBase, содержащая компоненты для работы с InterBase, минуя BDE. Эти компоненты обеспечивают повышенную производительность и позволяют использовать новые возможности сервера InterBase, недоступные обычным компонентам BDE. Этот новый вариант связи с базами данных будет рассмотрен в главе 10 в разделе 10.5.

9.2 Создание баз данных с помощью Database Desktop

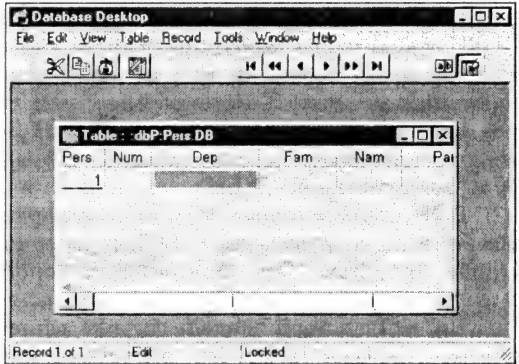
9.2.1 Создание новой таблицы

Прежде, чем начать строить приложения, работающие с базами данных, надо иметь сами базы данных. C++Builder поставляется с примерами, имеющими немало баз данных, которыми можно воспользоваться для обучения. Базы данных Pers и Der, фрагменты которых были приведены выше в таблицах 9.1 и 9.2 и которые будут использоваться в примерах данной и последующей глав книги, вы можете найти на диске, приложенном к этой книге. Но можно и самим создать необходимые базы данных. Причем не обязательно для этого использовать стандартные СУБД. Вместе с BDE и C++Builder поставляется программа Database Desktop (файл DBD.EXE для 16-разрядных приложений, файл DBD32.EXE для 32-разрядных приложений, файл DBDLOCAL.EXE — файл конфигурирования), которая позволяет создавать таблицы баз данных некоторых СУБД, задавать и изменять их структуру.

Обычно вызов Database Desktop включен в главное меню C++Builder в раздел Tools. Если это не сделано, то полезно включить его туда с помощью команды Tools | Configure Tools (см. раздел 14.2.4 главы 14). Вызовите Database Desktop. Вы увидите окно, показанное на рис. 9.4 (если в предыдущем сеансе работы с Database Desktop не была открыта какая-то таблица, то таблицы в середине окна не будет видно и разделов меню будет меньше).

Рис. 9.4.

Главное окно программы Database Desktop



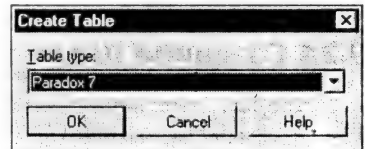
Давайте создадим с помощью Database Desktop таблицу базы данных СУБД Paradox 7. В Paradox 7 база данных — это каталог, в котором лежат таблицы — файлы с расширением **.db**. Поэтому прежде надо создать соответствующий каталог с помощью любой программы Windows, например, с помощью «Проводника». Далее выполните команду File | New в окне Database Desktop. Вам откроется подменю, содержащее три варианта:

QBE Query	Визуальный построитель запросов и запись этих запросов в файл
SQL File	Создание запроса на SQL и запись его в файл
Table	Создание новой таблицы

Выберите Table. Вам откроется небольшое диалоговое окно (рис. 9.5). В нем из выпадающего списка вы можете выбрать СУБД, для которой хотите создать таблицу. Выберите Paradox 7. Вы увидите окно, представленное на рис. 9.6. В этом окне вы можете задать структуру таблицы (поля и их типы), создать вторичные индексы, ввести диапазоны допустимых значений полей, значения по умолчанию и ввести много иной полезной информации о создаваемой таблице.

Рис. 9.5.

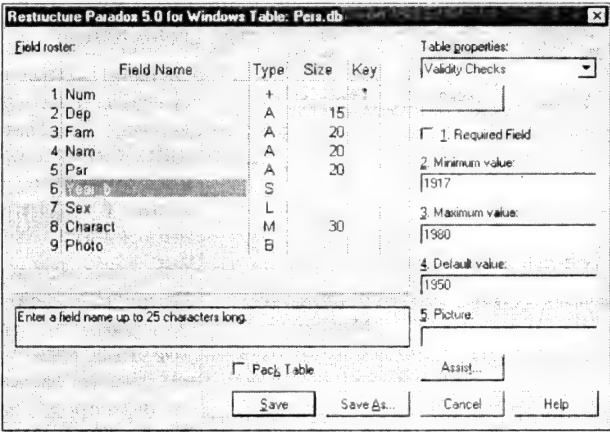
Окно выбора СУБД



9.2.2 Задание полей

Для каждого поля создаваемой таблицы прежде всего указывается имя (Field Name) — идентификатор поля. Он может включать до 25 символов и не может начинаться с пробела (но внутри пробелы допускаются). Затем надо выбрать тип (Type) данных этого поля. Для этого перейдите в раздел Type поля и щелкните правой кнопкой мыши. Появится список доступных типов, из которого вы можете выбрать необходимый вам. Приведем пояснения типов данных, используемых в Paradox.

Рис. 9.6.
Окно создания структуры таблицы
Paradox 7



Обозна- чение	Размер (Size)	Обозначение в списке	Пояснение
A	1 - 255	Alpha	Строковое поле, содержащее любые печатаемые ASCII символы. Размер — число символов.
N		Number	Действительные числа от -10 ³⁰⁷ до 10 ³⁰⁸ с 15 значащими разрядами. Для выбора формата представления надо использовать Paradox.
\$		Money	Положительные или отрицательные числа, отличающиеся от Number формой представления и символом денежной единицы. Для выбора формата представления надо использовать Paradox.
S		Short	Короткие целые числа от -32 767 до 32 767.
I		Long Integer	Длинные целые числа от -2 147 483 648 до 2 147 483 647.
#	0 - 32	BCD	Числа в формате BCD (Binary Coded Decimal). Вычисления с этими числами проводятся с повышенной точностью по сравнению с другими типами чисел, но медленнее. Этот тип введен для совместимости с другими приложениями, использующими BCD. В поле типа BCD можно вводить до 15 значащих разрядов.
D		Date	Значения, представляющие собой даты. Для выбора формата представления надо использовать Paradox.
T		Time	Значения, представляющие собой время. Для выбора формата представления надо использовать Paradox.
@		Timestamp	Значения, хранящие время и дату. Для выбора формата представления надо использовать Paradox. При вводе значения в поле типа Timestamp пользователь может последовательно нажимать клавишу пробела, чтобы ввести текущее время и дату.

Обозначение	Размер (Size)	Обозначение в списке	Пояснение
M	1 - 240	Memo	Поля для хранения текстов неограниченной длины. Тексты хранятся в отдельных файлах .mb . Указываемый размер — это число первых символов текста, хранящихся непосредственно в таблице. Просмотр полей Memo возможен в Paradox или в приложениях C++Builder.
F	0 - 240	Formatted Memo	Поля для хранения форматированных текстов неограниченной длины. Тексты хранятся в отдельных файлах .mb . Указываемый размер — это число первых символов текста, хранящихся непосредственно в таблице. Просмотр полей Formatted Memo возможен в Paradox или в приложениях C++Builder.
G		Graphic	Изображения из файлов в форматах .bmp , .pcx , .tif , .gif или .eps . Database Desktop преобразует их в формат .BMP . Просмотр полей Graphic возможен в Paradox или в приложениях C++Builder.
O		OLE	Данные типа OLE - изображения, звуки, документы. Database Desktop не поддерживает поля этого типа. Просмотр полей OLE возможен в Paradox или в приложениях C++Builder.
L		Logical	Логические поля. По умолчанию возможные значения — true и false . При вводе данных пользователь может ввести только первый символ из возможных значений.
+		Autoincrement	Автоматически увеличивающееся на 1 длинное целое. Только для чтения. При удалении записей значения полей в оставшихся записях не изменяются.
B		Binary	Данные, хранящиеся в отдельных двоичных файлах .mb , которые Database Desktop не интерпретирует. В файлах могут храниться звуки и любые другие данные.
Y	1 - 255	Bytes	Данные, которые Database Desktop не интерпретирует. В отличие от полей Binary хранятся в таблице, а не во внешних файлах.

В нашем примере таблицы Pers (см. таблицу 9.1) для поля **Num** целесообразно выбрать тип **Autoincrement**, обеспечивающий уникальность каждой записи. Для полей **Dep**, **Fam**, **Nam** и **Par** необходимо задать тип **Alpha**, для поля **Year_b** — тип **Short**, для поля **Sex** — **Logical**, для поля **Charact** — **Memo**, для поля **Photo** — **Graphic**. В таблице Dep (см. таблицу 9.2) для поля **Dep** надо задать тип **Alpha**, а для поля **Proisv** — **Logical**.

Для некоторых типов необходимо задавать размер (Size). Например, для строкового типа **Alpha** размер — это число символов.

Ключевые поля должны быть отмечены символом «*» в последней колонке. Для того, чтобы поставить или удалить этот символ, надо или сделать двойной щелчок в соответствующей графе информации о поле, или выделить эту графу и нажать клавишу пробела. Если имеется несколько ключевых полей, то в таблицах Paradox они должны быть первыми. В нашем примере для таблицы Pers ключевым является поле **Num**, а для таблицы Dep — поле **Dep**.

9.2.3 Задание свойств таблицы

Теперь обратите внимание на правую часть окна (рис. 9.6). В нем задаются свойства таблицы (Table properties). Вверху имеется выпадающий список с рядом разделов.

9.2.3.1 Validity Checks — проверка правильности значений

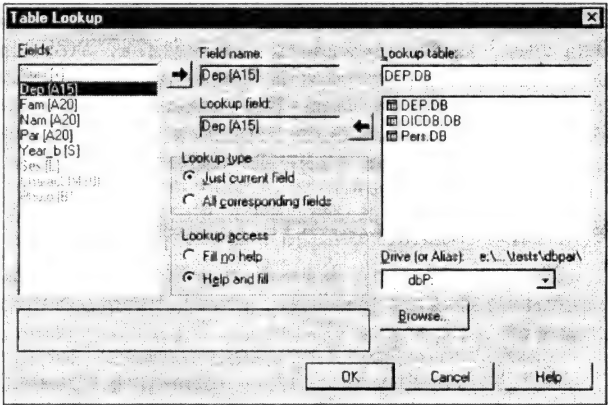
Начнем с первого из них: Validity Checks — проверка правильности значений. Вид правой части окна при выборе этого раздела показан на рис. 9.6 и может несколько изменяться в зависимости от того, какой тип у поля, выделенного курсором. Вы можете задать следующие характеристики поля:

Required Field	Этим индикатором отмечаются те поля, значения которых обязательно должны содержаться в каждой записи. Для нашего примера такими полями, вероятно, должны быть поля Fam , Nam и Par
Minimum	Минимальное значение. Это свойство полезно задавать для числовых полей. В нашем примере надо задать минимальное значение для поля Year_b
Maximum	Максимально значение. Это свойство полезно задавать для числовых полей. В нашем примере надо задать максимальное значение для поля Year_b
Default	Значение по умолчанию. Это свойство полезно задавать для числовых и логических полей. В нашем примере полезно задать значение по умолчанию для поля Year_b и обязательно надо задать значение по умолчанию для поля Sex (иначе у пользователя могут возникнуть проблемы при вводе информации)
Picture	Шаблон для ввода данных. Например, можно задать шаблон номера телефона «###-##-##» и др. Подробнее о составлении шаблонов вы можете узнать во встроенной справке Database Desktop
Assist	Эта кнопка вызывает диалоговое окно, помогающее создать шаблон Picture и занести его в список, из которого в дальнейшем его можно брать при создании новых таблиц

9.2.3.2 Table Lookup — таблица просмотра

Следующий раздел в выпадающем списке свойств таблицы в правом верхнем углу экрана на рис. 9.6: Table Lookup — таблица просмотра. Этот раздел позволяет связать с полем данной таблицы какое-то поле другой, просматриваемой таблицы, из которого будут браться допустимые значения. При выборе Table Lookup на экране появляется кнопка Define — определить. При ее нажатии открывается диалоговое окно, показанное на рис. 9.7. В нем вы можете для данного поля задать таблицу просмотра (lookup table). При этом вы можете воспользоваться выпадающим

Рис. 9.7.
Окно задания таблицы просмотра



списком драйверов или псевдонимов (Drive or Alias) и кнопкой просмотра (Browse). А затем кнопкой со стрелкой занести поле просматриваемой таблицы, из которого будут браться допустимые значения. В нашем примере пока все это сделать невозможно, поскольку еще не создана вторая таблица Dep. Однако, после того, как она будет создана, полезно вернуться к таблице Pers и для поля **Dep** задать таблицу Dep как просматриваемую и ее поле **Dep** как множество возможных значений. Это предотвратит ошибочное появление в таблице Pers каких-то значений подразделений, не содержащихся в таблице Dep.

9.2.3.3 Secondary Indexes — вторичные индексы

Следующий раздел в выпадающем списке свойств таблицы: Secondary Indexes — вторичные индексы. Этот раздел позволяет создать необходимые для дальнейшей работы вторичные индексы (первичный индекс создается по ключевым полям). Например, для дальнейшего использования нашей таблицы Pers полезны будут следующие индексы:

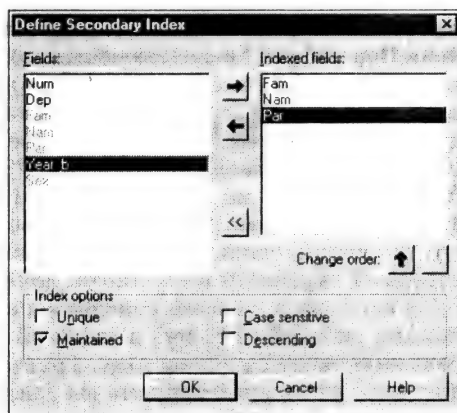
Имя индекса	Поля	Пояснение
fio	Fam, Nam, Par	Упорядочивание таблицы сотрудников по алфавиту.
depfio	Dep, Fam, Nam, Par	Упорядочивание таблицы по подразделениям, а внутри каждого подразделения упорядочивание сотрудников по алфавиту.
year	Year_b	Упорядочивание таблицы по году рождения сотрудников.

Чтобы создать новый вторичный индекс, нажмите кнопку Define — определить. Откроется диалоговое окно, представленное на рис. 9.8. В его левом окне Fields содержится список доступных полей, в правом окне Indexed fields вы можете подобрать и упорядочить список полей, включаемых в индекс. Для переноса поля из левого окна в правое надо выделить интересующее вас поле или группу полей и нажать кнопку со стрелкой вправо. Стрелками Change order (изменить последовательность) можно изменить порядок следования полей в индексе.

Панель радиокнопок Index Options (опции индекса) позволяют установить следующие характеристики:

Рис. 9.8.

Окно задания вторичного индекса

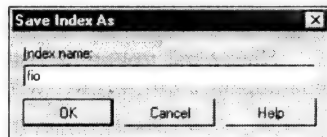


Unique	Установка этой опции не позволяет индексировать таблицу, если в ней находятся дубликаты совокупности включенных в индекс полей. Например, установка этой опции для индекса fio не допустила бы наличия в таблице сотрудников с совпадающими фамилией, именем и отчеством
Descending	При установке этой опции таблица будет упорядочиваться по степени убывания значений (по умолчанию упорядочивание производится по степени нарастания значений)
Case Sensitive	При установке этой опции будет приниматься во внимание регистр, в котором введены символы
Maintained	Если эта опция установлена, то индекс обновляется при каждом изменении в таблице. В противном случае индекс обновляется только в момент связывания с таблицей или передачи в нее запроса. Это несколько замедляет обработку запросов. Поэтому полезно включать эту опцию для обновляемых таблиц. Если таблица используется только для чтения, эту опцию лучше не включать

После того, как индекс сформирован, щелкаете на кнопке **OK**. Открывается окно (рис. 9.9), в котором вы задаете имя индекса.

Рис. 9.9.

Задание имени индекса



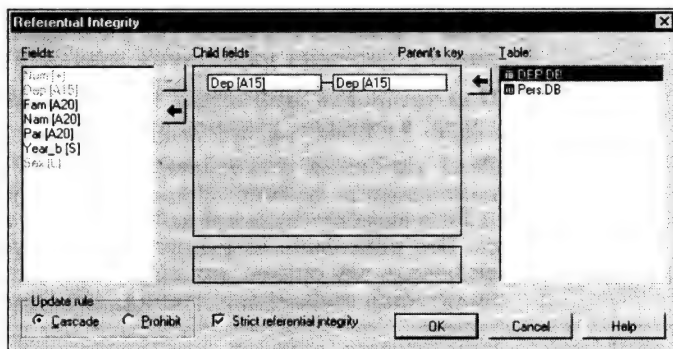
9.2.3.4 Referential Integrity — целостность на уровне ссылок

Следующий раздел в выпадающем списке свойств таблицы в правом верхнем углу экрана на рис. 9.6: **Referential Integrity** — целостность на уровне ссылок. Речь идет о способах, позволяющих обеспечить постоянные связи между данными отдельных таблиц. Если устанавливается целостность на уровне ссылок между двумя таблицами, одна из которых — головная (родительская), а другая — вспомогательная (дочерняя), то во вспомогательной таблице указывается поле или группа полей, которые могут брать свои значения только из ключевого поля (или полей) головной таблицы. Подобные связи допустимы не для всех типов таблиц, но в **Paradox** они предусмотрены. Прежде, чем создавать **Referential Integrity**, надо иметь обе связываемые таблицы — родительскую и дочернюю. Если бы мы уже имели в

нашем примере обе таблицы — Pers и Dep, мы могли бы задать целостность, связав поле **Dep** таблиц Pers с ключевым полем **Dep** головной таблицы Dep. Чтобы ввести подобную связь, надо сначала установить в качестве рабочего каталог, содержащий обе таблицы (это делается командой File | Working Directory). Затем надо открыть дочернюю таблицу Pers (команда File | Open), войти в режим ее реструктуризации (команда Table | Restructure) и в окне Table properties выбрать раздел Referential Integrity. Затем щелкнуть на кнопке Define, и перед вами откроется диалоговое окно, представленное на рис. 9.10. В его левой панели Fields вы можете выбрать поле или группу полей, связываемых с ключевыми полями головной таблицы, и кнопкой со стрелкой перенести их в список дочерних полей Child fields. Затем в правой панели Table вы можете указать головную панель (если ее там нет, значит вы неверно установили рабочий каталог) и кнопкой со стрелкой перенести в список ключей родительской таблицы Parent's key. Группа радиокнопок Update rule определяет, что будет, если в головной таблице вы удалите или измените значение ключевого поля, с которым связаны какие-то записи во вспомогательной таблице. Если вы установили опцию Prohibit, то Database Desktop просто не разрешит подобную операцию. Если же вы установили опцию Cascade, то при смене значения ключевого поля в головной таблице аналогичные изменения автоматически произойдут в записях дочерней таблицы. А если вы удалите запись в головной таблице, содержащую некоторое значение ключевого поля, то во вспомогательной таблице автоматически удалятся все записи, связанные с этим значением ключевого поля.

Рис. 9.10.

Окно установления целостности на уровне ссылок

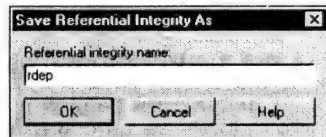


Установка индикатора Strict Referential Integrity в нижней части диалогового окна не позволит ранним версиям Paradox (в частности, версиям Paradox для DOS) открыть и испортить таблицы, в которых введена целостность на уровне ссылок.

Когда вы провели все необходимые операции, щелкните на кнопке OK и в открывшемся окне (рис. 9.11) введите имя созданной ссылки.

Рис. 9.11.

Ввод имени созданной ссылки



9.2.3.5 Password Security — пароли доступа

Следующий раздел в выпадающем списке свойств таблицы в правом верхнем углу экрана на рис. 9.6: Password Security — пароли доступа. Paradox позволяет задать для таблицы пароли и для каждого из них определить разрешенные операции как для таблицы в целом, так и для отдельных ее полей. Щелчок на кнопке Define откроет вам окно, показанное на рис. 9.12. В нем вы можете ввести главный пароль (окно Master password), подтвердить его (окно Verify master password), после чего

щелчком на кнопке Auxiliary Passwords (вспомогательные пароли) открыть новое диалоговое окно (рис. 9.13), позволяющее ввести вспомогательные пароли и определить правила доступа по ним.

Рис. 9.12.

Ввод главного пароля

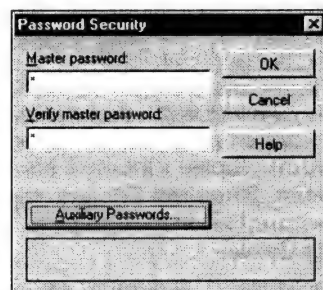
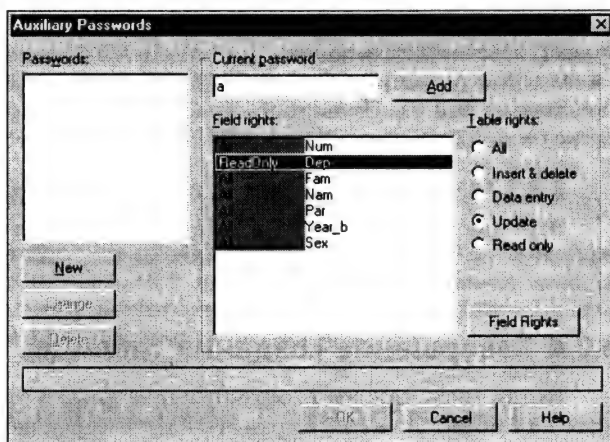


Рис. 9.13.

Ввод вспомогательного пароля



В окне Current Password (текущий пароль) вы указываете пароль (он совершенно не обязательно должен совпадать с тем, под которым вы вошли в это окно), для которого намереваетесь сформировать правила доступа. В группе радиокнопок Table Rights (права доступа к таблице) вы можете определить общий уровень доступа к таблице:

All	Допускаются любые операции, вплоть до изменения ее структуры, удаления таблицы, изменения и удаления паролей
Insert & Delete	Допускаются любые операции с записями (редактирование, вставка, удаление), но не разрешается изменение структуры таблицы и ее удаление
Data Entry	Допускается редактирование данных и вставка записей, но запрещено удаление записей и не разрешается изменение структуры таблицы и ее удаление
Update	Допускается только просмотр таблицы и изменение неключевых полей
Read Only	Допускается только просмотр таблицы

В окне Field Rights (права доступа к полю) вы можете определить дополнительные права доступа к каждому полю, но не превышающие заданный уровень доступа к таблице:

All	Дает все права доступа к полю, предусмотренные заданными правами доступа к таблице
Read Only	Позволяет только читать данные этого поля
None	Не позволяет ни наблюдать, ни редактировать данное поле

После того, как вы установили все права доступа для данного вспомогательно-го пароля, щелкните на кнопке Add и пароль занесется в окно списка паролей Pass-words. Далее кнопкой New вы можете начать задание нового вспомогательного па-роля. Кнопкой Change вы можете изменить выделенный в списке ранее введенный вспомогательный пароль или удалить его, щелкнув после кнопки Change на кноп-ке Delete.

9.2.3.6 Table Language — язык таблицы

Этот раздел в выпадающем списке Table Properties позволяет задать (если он не задан) или переопределить (кнопкой Modify) язык таблицы, установленный по умолчанию в драйвере данной СУБД с помощью программы BDE Administrator (см. раздел 9.3.3). Правильный выбор языка определяет, будут ли нормально чи-таться в таблице русские тексты.

9.2.3.7 Dependent Tables — зависимые таблицы

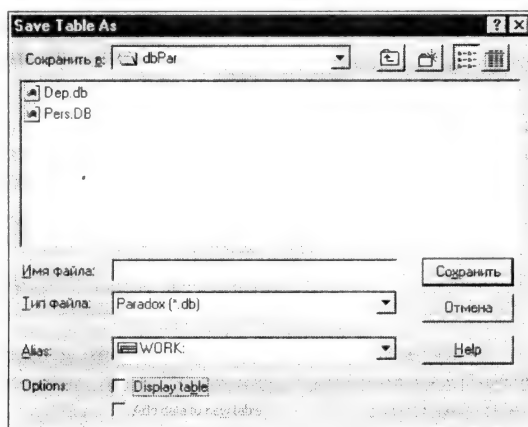
Этот последний раздел в выпадающем списке Table Properties позволяет про-смотреть список зависимых таблиц, связанных с данной целостностью на уровне ссылок Referential Integrity.

9.2.4 Завершение создания таблицы

После того, как все необходимые данные о структуре таблицы внесены, щелк-ните на кнопке Save as (сохранить как) и перед вами откроется окно (рис. 9.14), на-поминающее обычный диалог сохранения в файле. От обычного это окно отличает-ся выпадающим списком Alias. Этот список содержит псевдонимы различных баз данных (о них пойдет речь позднее), из которого вы можете выбрать базу данных, в которую будете сохранять свою таблицу. Если вам не надо сохранять таблицу в одной из существующих баз данных, то вы можете воспользоваться обычным спи-ском Сохранить в в верхней части окна. При этом вы с помощью обычной быстрой кнопки можете создать новую папку (каталог). Вспомните, что для Paradox база данных — это каталог, в котором сохраняется таблица.

Рис. 9.14.

Сохранение таблицы в базе данных



Внизу окна на рис. 9.14 имеются еще две опции. Первая из них — Display Table обеспечивает немедленное автоматическое открытие таблицы после ее сохранения. Вторая опция — Add Data to New Table доступна в случае, если производилось не создание таблицы, а изменение ее структуры. Эта опция обеспечивает, что в измененную структуру из прежней таблицы перенесутся все данные, которые вписываются в новую структуру.

Мы рассмотрели создание таблицы Paradox. Для других СУБД диалоги отличаются от рассмотренного и учитывают возможности различных СУБД. Однако эти отличия касаются только отдельных деталей и рассматривать их мы не будем.

9.2.5 Изменение структуры и заполнение таблицы с помощью Database Desktop

После того, как вы создали таблицу, вы можете ее открыть командой File | Open. Впрочем, если при сохранении структуры таблицы вы использовали описанную выше опцию Display Table, то таблица откроется автоматически. В обоих случаях вы увидите окно вида, представленного ранее на рис. 9.4. С помощью разделов меню Table вы можете смотреть содержимое таблицы (команда Table | View Data) или редактировать его (команда Table | Edit Data). Впрочем, вряд ли это целесообразно делать. Программа Database Desktop не настраивается на русский язык, так что все, вводимое русскими буквами, выглядит абракадаброй. Впрочем, в дальнейшем при использовании такой таблицы в приложении все надписи будут выглядеть нормально.

Команда Table | Info Structure позволяет просмотреть информацию о структуре таблицы, а команда Table | Restructure позволяет изменить структуру таблицы или какие-то ее характеристики. При выполнении этой команды вы попадаете в окно, аналогичное используемому ранее при разработке структуры.

9.3 Создание и редактирование псевдонимов баз данных, каталогов, драйверов

Имеется три альтернативных пути просмотра, создания и редактирования псевдонимов с помощью трех различных программ: Database Desktop, BDE Administrator и Database Explorer. Рассмотрим их все, наиболее подробно остановившись на Database Desktop.

9.3.1 Автоматически создаваемые псевдонимы рабочего и частного каталогов

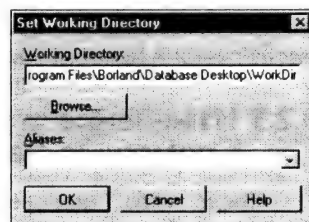
Уже говорилось (см. раздел 9.1.1) о значении присваивания псевдонимов базам данных и каталогам. Но прежде, чем говорить о создании новых псевдонимов, надо упомянуть о двух псевдонимах, автоматически создаваемых BDE. Эти псевдонимы относятся к двум каталогам: *рабочему* (working) и *частному* (private).

Рабочий каталог используется для совместной работы всех пользователей. Database Desktop создает его в момент установки в своем рабочем каталоге с путем ...\\Program Files\\Borland\\Database Desktop\\WorkDir. Он имеет псевдоним WORK. Изменить рабочий каталог можно с помощью Database Desktop, выполнив команду File | Working Directory. Откроется окно, приведенное на рис. 9.15. В нем вы можете задать новый рабочий каталог (Working Directory), или найти его поиском по кнопке Browse, или выбором из выпадающего списка Aliases — псевдонимы. При смене рабочего каталога псевдоним WORK автоматически будет подразумевать этот новый каталог. Если вы — единственный или основной пользователь Database Desktop, то

полезно в качестве рабочего установить тот каталог, внутри которого или в подкаталогах которого сосредоточено большинство ваших баз данных. Это сократит время на открытие таблиц и другие операции, которые предлагают в качестве каталога прежде всего псевдоним WORK. Кроме того, как будет сказано ниже, полезно изменять рабочий каталог, чтобы иметь доступ к своим файлам конфигурации Database Desktop.

Рис. 9.15.

Установка псевдонима рабочего каталога



Личный (private) каталог также создается автоматически при установке Database Desktop там же, где и рабочий каталог. Он служит для хранения личных таблиц и других объектов пользователя, не предназначенных для всеобщего обозрения. Он имеет псевдоним PRIV. Изменить частный каталог можно с помощью Database Desktop, выполнив команду File | Private Directory. Откроется окно, аналогичное приведенному на рис. 9.15. В нем вы можете задать новый личный каталог. При этом псевдоним PRIV автоматически будет подразумевать этот новый каталог.

9.3.2 Создание и просмотр псевдонимов баз данных в Database Desktop

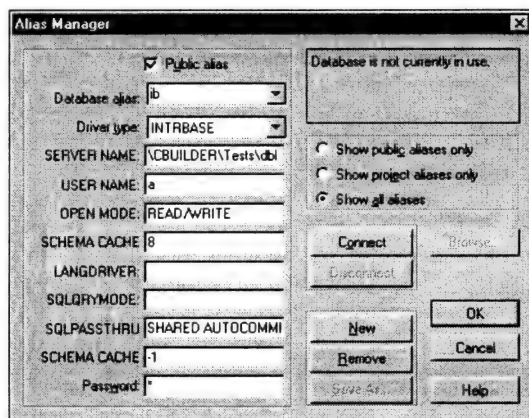
Теперь обсудим создание новых псевдонимов баз данных. Могут создаваться псевдонимы двух видов: *открытые*, доступные при работе из любого каталога, и *псевдонимы проекта*, доступные только при работе в конкретном рабочем каталоге. Открытые псевдонимы сохраняются в каталоге ...\Borland\Borland Shared\Bde в файле IDAPI.CFG в C++Builder 5 или в файле IDAPI32.CFG в ранних версиях C++Builder. Они доступны из любого рабочего каталога. Псевдонимы проекта сохраняются в файле IDAPI.CFG в рабочем каталоге и доступны только из этого каталога. Так что вы можете в разных каталогах разместить файлы конфигурации с разными псевдонимами проекта и в зависимости от того, какой каталог вы назначите рабочим, будете иметь разные наборы псевдонимов.

Псевдонимы можно просматривать и создавать в Database Desktop, выполнив команду Tools | Alias Manager. Вы увидите диалоговое окно Alias Manager (диспетчера псевдонимов), вид которого представлен на рис. 9.16. Впрочем, вид этого окна существенно зависит от того, псевдоним какой базы данных просматривается. Индикатор Public alias (открытый псевдоним) в верхней части окна показывает, будет ли создаваться открытый псевдоним, или псевдоним проекта. Ниже расположен выпадающий список Database Alias, в котором вы можете выбрать интересующий вас псевдоним из числа уже созданных. То, какие именно псевдонимы в нем видны, определяется группой радиокнопок справа. Если выбрана кнопка Show Public Aliases Only, то в списке отображаются только открытые псевдонимы; если выбрана кнопка Show Project Aliases Only, то отображаются только псевдонимы проекта; при выбранной кнопке Show All Aliases отображаются псевдонимы обоих типов.

При выборе псевдонима в списке Database Alias автоматически изменяется тип драйвера в выпадающем списке Driver type и расположенная ниже информация о драйвере. На рис. 9.16 изображен случай базы данных INTERBASE с драйвером INTRBASE (аналогично выглядит экран и для любого драйвера SQL Link — ORACLE, SYBASE). В информации указывается:

Рис. 9.16.

Просмотр псевдонимов баз данных



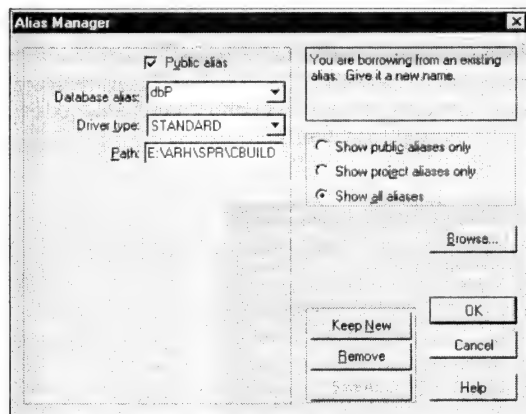
SERVER NAME имя сервера	Задается путь и файл базы данных или имя сервера
USER NAME имя пользователя	Задается имя пользователя
OPEN MODE режим открытия	Возможные значения: READ/WRITE (для чтения и записи) или READ ONLY (только для чтения)
SCHEMA CACHE SIZE размер кэша схем	Определяет число кэшируемых схем информации. По умолчанию — 5, может выбираться от 0 до 32
LANGDRIVER драйвер языка	Определяет множество отображаемых символов
SQLQRYMODE режим обработки запросов SQL	Может принимать значения: NULL (по умолчанию) — обращение сначала к серверу, а если он не может обработать запрос, то к Desktop, SERVER — только к серверу и, если он не может обработать запрос, то отказ, LOCAL — только к Desktop
SQLPASSTHRU MODE доступ к QBE и SQL	Определяет, может ли пользователь Database Desktop иметь доступ к QBE и SQL редакторам в пределах одного соединения. Возможные значения: NOT SHARED — не может, SHARED AUTOCOMMIT (по умолчанию) может и результаты запроса SQL автоматически фиксируются в базе данных, SHARED NO AUTOCOMMIT — может, но результаты запроса SQL автоматически не фиксируются в базе данных
Password пароль	Пароль может задаваться, а может и не задаваться

Кнопка Connect (соединение) позволяет немедленно соединиться с базой данных. Кнопка Remove (удаление) позволяет удалить псевдоним. Кнопка Save as позволяет сохранить список псевдонимов (если он изменялся) в файлах конфигурации заданного каталога. А кнопка New (новый) позволяет создать новый псевдоним.

При щелчке на этой кнопке диалоговое окно несколько преобразуется (рис. 9.17). Правда, различие рис. 9.16 и рис. 9.17 объясняется не только этим, но и выбором в этих двух примерах разных драйверов. Основное же отличие — изменение кнопок (в частности, New меняется на Keep New — сохранить новый псевдоним) и возможность в списке Database Alias не только выбирать существующий псевдоним, но и писать новый.

Рис. 9.17.

Задание нового псевдонима базы данных



В открывшемся окне надо:

- Установить или убрать опцию Public alias для создаваемого псевдонима (ее смысл рассматривался выше).
- Выбрать драйвер базы данных в списке Driver type и заполнить его характеристики. Для баз данных Paradox, dBase и ряда других надо выбрать тип драйвера STANDARD, в котором достаточно указать каталог хранения таблиц. В этом может помочь кнопка просмотра Browse. Для SQL-драйверов заполняемые позиции были рассмотрены выше. В ряде случаев проще выбрать сначала в окне Database Alias какой-то из имеющихся псевдонимов, с драйвером и характеристиками, подобными нужным для нового псевдонима, а затем отредактировать эти характеристики.
- В окне Database Alias написать новый псевдоним.
- Щелкнуть на кнопке Keep New, чтобы сохранить введенную информацию и перейти к созданию следующего псевдонима, или сохранить всю информацию кнопкой Save as и выйти из диалога.

9.3.3 Создание и просмотр псевдонимов драйверов и баз данных в BDE Administrator

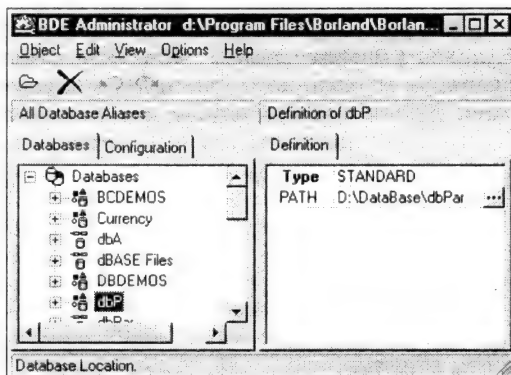
Программа BDE Administrator (Администратор BDE) позволяет просматривать, редактировать и создавать новые драйверы BDE баз данных различных типов: стандартные (STANDARD), SQL, Access, ODBC. При установке BDE на компьютер программа BDE Administrator включается в состав программы «Панель управления» Windows. Вы можете также включить ее в меню C++Builder с помощью команды Tools | Configure Tools. Файл программы — ... \Program Files\Borland\Borland Shared\BDE\bdeadmin.exe.

Окно программы (рис. 9.18) имеет две страницы: Databases — базы данных и Configuration — конфигурация. На странице в левой панели расположено дерево псевдонимов баз данных. Выделив интересующий вас псевдоним в левой панели, вы можете в правой панели Definition увидеть все его характеристики. Число и смысл этих характеристик зависит от используемого драйвера.

Для драйвера STANDARD, используемого, в частности, для баз данных Paradox, набор характеристик псевдонима минимальный: Type — имя драйвера и PATH — путь к базе данных. Щелкнув на параметре PATH, вы увидите кнопку с многоточием (рис. 9.18). При ее нажатии отобразится обычный диалог Windows, позволяющий выбрать новый каталог. Таким образом вы можете изменить характеристики псевдонима, если, например, изменилось расположение базы данных на

Рис. 9.18.

Страница Databases Администратора BDE



диске. После этого все приложения, использующие этот псевдоним, автоматически будут работать с данными, даже не заметив изменения их месторасположения.

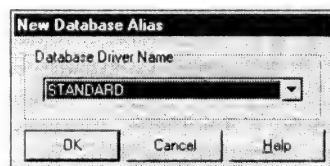
Характеристики драйверов баз данных INTERBASE и драйверов SQL Link (ORACLE, SYBASE и др.) были рассмотрены выше.

Имейте в виду, что в правой панели можно изменять только те параметры, имена которых не выделены жирным шрифтом. Значения выделенных параметров изменять нельзя.

Для создания нового псевдонима надо щелкнуть правой кнопкой мыши и во всплывшем меню выбрать раздел New — новый псевдоним. Перед вами появится небольшое диалоговое окно, показанное на рис. 9.19. В его выпадающем списке вы должны выбрать драйвер для создаваемого псевдонима. Тип драйвера STANDARD можно использовать для таблиц Paradox, dBASE, FoxPro, для текстов ASCII.

Рис. 9.19.

Диалоговое окно выбора драйвера для нового псевдонима



Выбрав драйвер, щелкните на ОК и в дереве псевдонимов появится новая вершина, для которой вы можете задать имя — псевдоним.

В правой панели появятся параметры, которые вы должны установить для создаваемого псевдонима. Для типа STANDARD эти параметры следующие:

PATH	Путь к базе данных
DEFAULT DRIVER	Один из следующих драйверов: PARADOX для таблиц Paradox (файлов .db), DBASE для таблиц dBASE (файлов .dbf), FOXPRO для таблиц FoxPro (файлов .dbf), ASCIIDRV для текстов ASCII (файлов .txt)
ENABLE BCD	Определяет, должна ли BDE транслировать числовые поля в значения с плавающей запятой, или в коды BCD. BCD позволяет избежать ошибок округления. Если ENABLE BCD = true , то поля DECIMAL и NUMERIC преобразуются в коды BCD

После задания параметров щелкните правой кнопкой мыши и из всплывшего меню выберите раздел Apply. Новый псевдоним будет зафиксирован.

Для удаления существующего псевдонима выделите его в левой панели, щелкните правой кнопкой мыши и из всплывшего меню выберите раздел Delete.

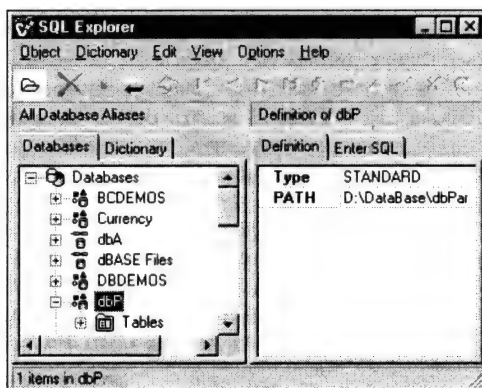
Мы рассмотрели страницу Databases окна Администратора BDE. Страница Configuration (конфигурация) позволяет изменить какие-то из параметров выбранного драйвера и установить системные параметры, определяющие, как преобразовываются строки с обозначением дат, времени и чисел в соответствующие значения.

9.3.4 Создание и просмотр псевдонимов в SQL Explorer

Вызов этой программы осуществляется из главного меню C++Builder командой Database | Explore. Окно программы SQL Explorer представлено на рис. 9.20. В его левой панели на странице Databases вы можете видеть дерево зарегистрированных псевдонимов. Выделив интересующий вас псевдоним (на рис. 9.20 это псевдоним **dbP**), вы увидите на правой панели его параметры. При желании вы можете их изменить.

Рис. 9.20.

Окно программы SQL Explorer — просмотр параметров псевдонима

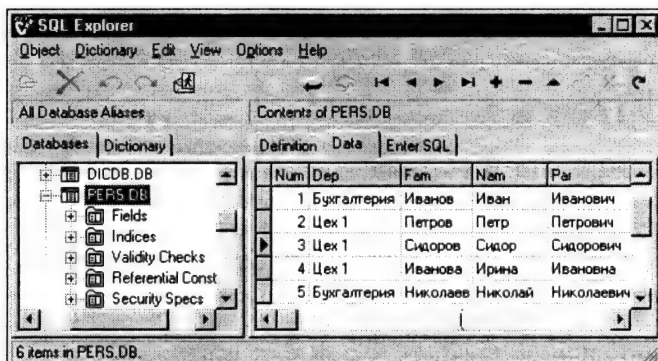


Выделив корневой узел в левой панели, вы можете с помощью команды Object | New создать новый псевдоним. При этом вначале вам будет показано окно выбора драйвера, приведенное ранее на рис. 9.19 при рассмотрении работы с Администратором BDE. Все дальнейшие действия не отличаются от тех, которые необходимы при создании псевдонима с помощью Администратора BDE. Так что посмотрите раздел 9.3.3, в котором вся эта процедура описана достаточно подробно.

Однако возможности SQL Explorer по просмотру и модификации баз данных не ограничиваются модификацией существующих и созданием новых псевдонимов. Вы можете выбрать в левой панели интересующий вас псевдоним базы данных, и просмотреть, что хранится в этой базе данных (таблицы, индексы и т.п.), просмотреть таблицы (этот момент отображен на рис. 9.21). При этом в правой панели на странице Definition вы можете просмотреть общую информацию о таблице,

Рис. 9.21.

Окно программы SQL Explorer — просмотр данных в таблице Pers

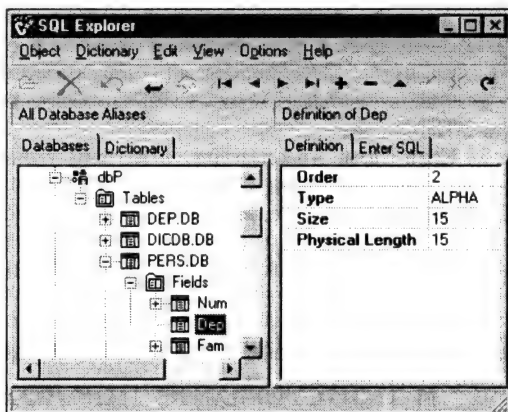


на странице Text вы можете увидеть текст запроса SQL, создавшего данную таблицу (если использовался драйвер SQL). На странице Data вы можете просмотреть хранящуюся в таблице информацию. При этом кнопки навигатора сверху диалогового окна позволяют вам редактировать записи, вставлять новые записи и т.п. На странице Enter SQL можете сформировать и выполнить запрос к таблице.

Вы можете продвинуться далее и раскрыть структуру таблицы. На рис. 9.22 изображен просмотр объявления столбца **Dep** таблицы Pers, причем вы можете изменить соответствующую информацию.

Рис. 9.22.

Окно программы SQL Explorer — просмотр объявления поля Dep



Программа SQL Explorer имеет еще много возможностей, которые будут рассмотрены в дальнейшем. А завершая эту главу, хотелось бы, чтобы вы создали с помощью одного из рассмотренных инструментов базу данных с псевдонимом **dbP**, содержащую две таблицы Pers и Dep, описанные в разделе 9.1 в таблицах 9.1 и 9.2. Это необходимо для работы с примерами приложений, которые мы будем рассматривать в последующих разделах.

9.4 Обзор компонентов, используемых для связи с базами данных

Компоненты, используемые для работы с базами данных, расположены в библиотеке компонентов на страницах Data Access (доступ к данным) и Data Control (управление данными).

Каждое приложение, использующее базы данных, обычно имеет по крайней мере по одному компоненту следующих трех типов:

- Компоненты — наборы данных (data set), непосредственно связывающиеся с базой данных. Это такие компоненты, как **Table**, **Query**, **StoredProc**.
- Компонент — источник данных (data source), осуществляющий обмен информацией между компонентами первого типа и компонентами визуализации и управления данными. Таким компонентом является **DataSource**.
- Компоненты визуализации и управления данными, такие, как **DBGrid**, **DBText**, **DBEdit** и множество других.

Связь этих компонентов друг с другом и с базой данных можно представить схемой, приведенной на рис. 9.23.

Помимо указанных компонентов в приложении может размещаться компонент **Database**. Этот компонент в основном используется в приложениях, работающих на платформе клиент/сервер. Он обеспечивает общение с удаленным сервером.

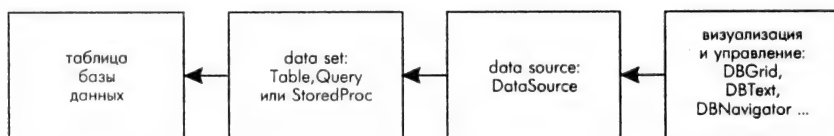


Рис. 9.23.

Схема взаимодействия компонентов C++Builder с базой данных

ром, реализацию транзакций, работу с паролями. Компонент **Database** целесообразно вводить в приложение только в сравнительно редких случаях. Если он не введен явно, C++Builder автоматически создает его для каждой используемой в приложении базы данных. Подробнее этот компонент будет рассмотрен в разделе 10.2.2 главы 10.

Еще один компонент, который тоже автоматически создается C++Builder — компонент **Session**. Это главный компонент любого приложения, работающего с базами данных. Но в явном виде эти компоненты имеет смысл вводить только в многозадачные приложения, в которых параллельно обрабатывается несколько потоков информации. Компонент **Session** будет рассмотрен в разделе 9.8.

9.5 Основные свойства компонента Table и простейшие приложения на его основе

9.5.1 Установка связей между компонентами и базой данных, навигация по таблице

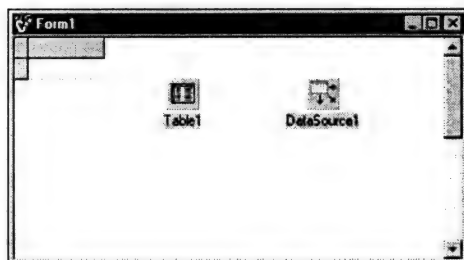
Давайте построим простейшее приложение, работающее с базой данных. Будем использовать ту таблицу **Paradox Pers**, которую вы создавали ранее в базе данных с псевдонимом **dbP**. Она имеется на прилагаемом к книге диске.

В нашем простейшем приложении мы будем в качестве набора данных использовать компонент **Table**. Откройте новое приложение и перенесите на форму компонент **Table** со страницы библиотеки **Data Access**. Перенесите также на форму с той же страницы библиотеки компонент **DataSource**, который будет являться источником данных. Оба эти компонента невидимые, пользователю они будут не видны, так что их можно разместить в любом месте формы. В качестве компонента визуализации данных возьмите компонент **DBGrid** со страницы **Data Control**. Это визуальный компонент, в котором будут отображаться данные таблицы. Поэтому растяните его пошире, или можете в его свойстве **Align** установить **alClient** (рис. 9.24).

Теперь нам надо установить цепочку связей между этими компонентами, показанную выше на рис. 9.23. Главное свойство **DBGrid** и других компонентов визуализации и управления данными — **DataSource**. Выделите на форме компонент **DBGrid1** и щелкните на его свойстве **DataSource** в Инспекторе Объектов. Вы увидите выпадающий список, в котором перечислены все имеющиеся на форме источ-

Рис. 9.24.

Форма простого приложения, работающего с базой данных



ники данных. В нашем случае имеется только один источник данных — **DataSource1**. Установите его в качестве значения свойства **DataSource**. Далее надо установить связь между источником данных и набором данных. Выделите компонент **DataSource1** и найдите в Инспекторе Объектов его главное свойство — **DataSet**. Щелкните на этом свойстве и из выпадающего списка выберите **Table1** (если бы у вас было несколько компонентов — наборов данных, то все они были бы в этом списке).

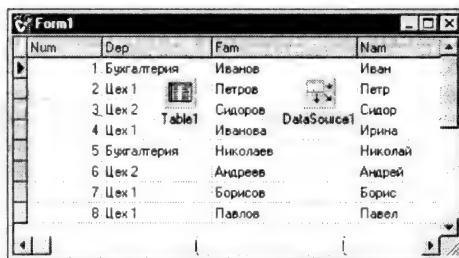
Теперь осталось связать компонент **Table1** с необходимой таблицей базы данных. Для этого служат два свойства компонента **Table**: **DatabaseName** и **TableName**. Прежде всего надо установить свойство **DatabaseName**. В выпадающем списке этого свойства в Инспекторе Объектов вы можете видеть все доступные BDE псевдонимы баз данных. Выберите из этого списка псевдоним **dbP**, который вы сами ввели ранее. Если этого псевдонима там нет, значит вы забыли его создать. В этом случае создайте его с помощью, например, Database Desktop, как описано в разделе 9.3.2.

После этого можно устанавливать значение свойства **TableName**. В выпадающем списке этого свойства перечислены таблицы, доступные в данной базе данных. Выберите таблицу **Pers**.

А теперь наступает самый ответственный момент. Вы можете прямо в процессе проектирования соединиться с базой данных. Соединение осуществляется свойством **Active**. По умолчанию оно равно **false**. Установите его в **true**. Если все сделано вами правильно, то вы увидите в поле компонента **DBGrid1** данные из таблицы (рис. 9.25) и сможете просмотреть их.

Рис. 9.25.

Форма простого приложения после соединения компонента **Table** с базой данных



Следует сразу отметить, что заранее выставлять для таблиц **Active = true** допустимо только в процессе настройки и отладки приложения, работающего с локальными базами данных.

Хороший стиль программирования

В законченном приложении во всех таблицах сначала должно быть установлено **Active = false**, затем при событии формы **OnCreate** эти свойства могут быть установлены в **true**, а при событии формы **OnDestroy** эти свойства опять должны быть установлены в **false**. Это исключит неоправданное поддержание связи с базой данных, которое занимает ресурсы, а при работе в сети мешает доступу к базе данных других пользователей.

Вы можете прямо сейчас, хотя приложение еще не закончено, сохранить проект, запустить его на выполнение и убедиться, что с ним можно работать. Вы можете просматривать данные, редактировать их (редактируемые данные будут помещаться в базу данных в момент перехода от редактируемой записи к любой другой). Вы можете также убедиться, что в таблице **Pers** базы данных **dbP** невозможно изменить поле **Num**, поскольку оно автоматически изменяется и доступно только для чтения (см. раздел 9.2.2). Вы увидите также, что нельзя задать произвольное имя подразделения **Dep**, поскольку имеется таблица просмотра **Dep**, заданная целостность на уровне ссылок и допустимы только те значения **Dep**, которые имеются в головной таблице **Dep** (см. разделы 9.2.3.2 и 9.2.3.4).

При попытке делать подобные неразрешенные исправления соответствующие сообщения об ошибках будут появляться не в момент редактирования, а после редактирования при попытке перейти к следующей записи, поскольку именно в этот момент отредактированная запись должна заноситься в таблицу. Правда, если вы хотите проверять все эти возможности, то надо бы отключить остановы при генерации исключений. Для этого надо выполнить команду **Tools | Debugger Options**, в открывшемся диалоговом окне выбрать страницу **Language Exceptions** и выключить опции **Stop on Delphi Exceptions** и **Stop On C++ Exceptions** (см. раздел 14.2.8).

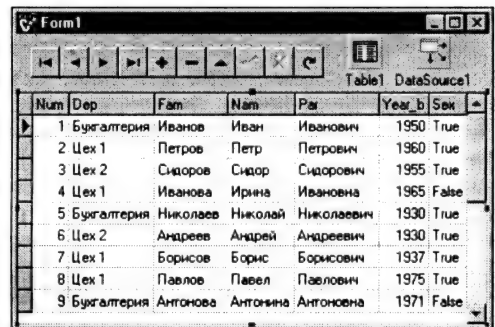
Разрешить пользователю так просто, как в созданном вами приложении, редактировать данные в таблице в большинстве случаев недопустимо. Слишком велика вероятность, что без соответствующей проверки вводимых данных будут сделаны ошибки. Предотвратить редактирование данных вы можете, установив свойство **ReadOnly** компонента **DBGrid1** в **true**. Другой способ сделать то же самое — установить в свойстве **Options** подсвойство **dgEditing** в **false**. Попробуйте оба варианта и вы увидите некоторое различие между ними во время выполнения. В свойстве **Options** есть еще много полезных подсвойств. Посмотрите сведения о них во встроенной справке **C++Builder**.

Отметим еще одно свойство компонента **Table** — **Exclusive**. Это свойство определяет доступ к используемой таблице при одновременном обращении к ней нескольких приложений (например, при работе в сети или в многозадачном режиме). Если задать значение этого свойства **true**, то таблица будет закрыта для других приложений. Свойство можно изменять только при **Active = false**.

В спроектированное вами простейшее приложение можно добавить еще один компонент, управляющий работой с таблицей — навигатор **DBNavigator**, расположенный на странице **Data Control** библиотеки компонентов. Измените свойство **Align** компонента **DBGrid1** на **alBottom**, сдвиньте верхний край этого компонента немного вниз и на верх формы поместите компонент **DBNavigator** (см. рис. 9.26).

Рис. 9.26.

Форма простого приложения с навигатором



Компонент имеет ряд кнопок, служащих для управления данными. Перечислим их названия и назначение, начиная с левой кнопки:

nbFirst	перемещение к первой записи
nbPrior	перемещение к предыдущей записи
nbNext	перемещение к следующей записи
nbLast	перемещение к последней записи
nbInsert	вставить новую запись перед текущей
nbDelete	удалить текущую запись
nbEdit	редактировать текущую запись

nbPost	послать отредактированную информацию в базу данных
nbCancel	отменить результаты редактирования или добавления новой записи
nbRefresh	очистить буфер, связанный с набором данных

Пользуясь свойством навигатора **VisibleButtons**, можно убрать любые ненужные в данном приложении кнопки. Например, если вы не хотите разрешить пользователю вводить в базу данных новые записи, то можете установить в **false** кнопку **nbInsert**. Если вы хотите вообще запретить редактирование, то можно оставить только кнопки **nbFirst**, **nbPrior**, **nbNext** и **nbLast**, а все остальные убрать.

Чтобы приложение с навигатором работало, надо установить основное свойство навигатора — **DataSource** — источник данных (имя компонента **DataSource**).

Откомпилируйте приложение, выполните его и посмотрите в работе.

9.5.2 Свойства полей

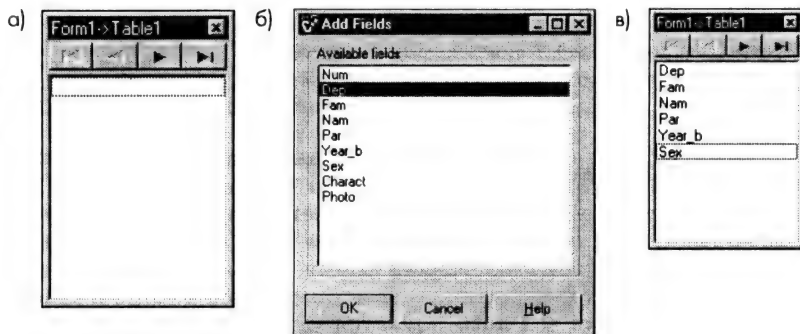
Наше приложение выглядит, конечно, очень плохо. Во-первых, последовательность записей определяется ключевым полем **Num**, а хотелось бы, чтобы записи были расположены по алфавиту или по отделам и алфавиту. Первое поле с номерами записей вообще пользователю не нужно и надо бы, чтобы его не было видно. Шапка таблицы содержит непонятные пользователю имена полей **Num**, **Fam** и т.д., а надо, чтобы были написаны нормальные заголовки по-русски. В графе **Sex** значения **true** и **false**, а нужны нормальные обозначения типа «м», «ж» или «мужской», «женский».

Все это можно легко поправить. Начнем с упорядочивания записей. Выделите на форме компонент **Table1**. В Инспекторе Объектов вы увидите среди прочих свойства **IndexName** и **IndexFieldName**. Первое из них содержит выпадающий список индексов, созданных для вашей таблицы. Выберите, например, индекс **fio**, и увидите, что записи окажутся упорядоченными по алфавиту, поскольку в этот индекс включены поля **Fam**, **Nam** и **Par**. При индексе **defio** упорядочивание будет по подразделениям, а внутри каждого подразделения — по алфавиту. Альтернативный вариант индексации предоставляет свойство **IndexFieldName**. В нем просто перечислены предусмотренные комбинации полей и вы можете выбрать необходимую, если забыли, что обозначают имена индексов.

Теперь займемся отдельными полями. Для их редактирования служит Редактор Полей. Вызвать его проще всего двойным щелчком на компоненте **Table1**. Сначала вы увидите пустое поле этого редактора (рис. 9.27 а). Щелкните на нем правой кнопкой мыши и из всплывающего меню выберите раздел **Add fields** (добавить поля). Вы увидите окно, изображенное на рис. 9.27 б, в котором содержится список всех полей таблицы. Выберите из него курсором мыши интересующие вас поля. Если вы при этом будете держать нажатой клавишу **Ctrl**, то может выделить

Рис. 9.27.

Редактор Полей:
исходное состояние
(а), окно выбора
полей (б), состояние
после выбора (в)



любую комбинацию полей. Однако, имейте в виду, что только к тем полям, которые вы добавите, вы сможете в дальнейшем обращаться. Так что в данном случае вам имеет смысл выделить все поля, кроме **Charact**, **Photo** и, может быть, **Num**. Выделив поля, щелкните на **OK** и вы вернетесь к основному окну Редактора Полей, но в нем уже будет содержаться список добавленных полей (рис. 9.27).

Эти поля будут соответствовать колонкам таблицы. Изменить последовательность их расположения можно, перетаскивая мышью идентификатор какого-то поля на нужное место. Как мы увидим далее, те поля, которые не должны отображаться в таблице, могут быть сделаны невидимыми.

Выделите в списке какое-то поле и посмотрите его свойства в Инспекторе Объектов. Вы увидите, что каждое поле — это объект, причем его класс зависит от типа поля: **TStringField**, **TSmallintField**, **TBooleanField** и т.п. Все эти классы являются производными от **Tfield** — базового класса полей. Таким образом, каждое поле является объектом и обладает множеством свойств. Рассмотрим основные из них, которые чаще всего необходимо задавать.

Свойство **Alignment** определяет выравнивание отображаемого текста внутри колонки таблицы: влево, вправо или по центру.

Свойство **DisplayLabel** соответствует заголовку столбца данного поля. Например, для поля **Fam** значение **DisplayLabel** можно задать равным «Фамилия», для **Nam** — «Имя» и т.д.

Свойство **DisplayWidth** определяет ширину колонки — число символов.

Свойства **EditMask** для строк и **EditFormat** для чисел определяют форматы отображения данных.

Для логических полей (в нашем примере для поля **Sex**) очень важным свойством является **DisplayValues**. Это свойство определяет, какие значения должны отображаться, если поле имеет значение **true** или **false**. Отображаемые значения разделяются точкой с запятой. Первым пишется значение, соответствующее **true**. Например: «м;ж» или «мужской;женский».

Свойство **ReadOnly**, установленное в **true**, запрещает пользователю вводить в данное поле значения. Свойство **Visible** определяет, будет ли видно пользователю соответствующее поле. В нашем примере, вероятно, можно задать **Visible = false** для поля **Num**, если вы его не исключили из списка полей **Table1**.

После установки всех необходимых свойств приложение уже приобретает приемлемый вид (рис. 9.28).

Рис. 9.28.

Приложение с установленными свойствами полей и удаленными кнопками редактирования в навигаторе

Отдел	Фамилия	Имя	Отчество	г.р.	Пол
Бухгалтерия	Антонова	Антонина	Антоновна	1955	ж
Бухгалтерия	Иванов	Иван	Иванович	1950	м
Бухгалтерия	Николаев	Николай	Николаевич	1930	м
Цех 1	Борисов	Борис	Борисович	1937	м
Цех 1	Иванова	Ирина	Ивановна	1961	ж
Цех 1	Павлов	Павел	Павлович	1975	м
Цех 1	Петров	Петр	Петрович	1960	м
Цех 2	Андреев	Андрей	Андреевич	1930	м
Цех 2	Иванников	Иван	Иванович	1975	м

Отметим еще одну особенность Редактора Полей — возможность перетаскивать из него поля на форму с помощью мыши. Захватите в Редакторе Полей какое-нибудь поле, например, **Fam**, и перетащите его на форму. На форме появится связанный с этим полем компонент типа **TDBEdit** и метка, содержащая его свойство **DisplayLabel**. При переносе на форму булева поля **Sex** на ней появится связанный с этим полем индикатор — компонент типа **TDBCheckBox**. При перетаскивании полей **Charact** и **Photo** появятся соответственно связанные с этими полями

компоненты типов **TDBMemo** и **TDBImage**. И во всех этих компонентах, если вы переключите свойство **Active** компонента **TTable** в **true**, сразу отобразятся соответствующие данные из таблицы.

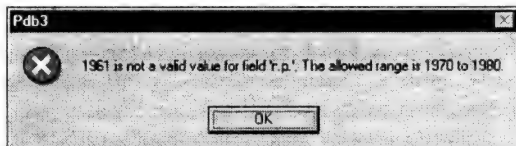
9.5.3 Ограничения вводимых значений

Ранее мы обсуждали вопрос об ограничении диапазона возможных значений полей при создании таблиц. Однако, **C++Builder** предусматривает еще разнообразные возможности дополнительного ограничения значений в приложении. Дело в том, что в таблице устанавливаются обычно достаточно широкие пределы, пригодные для любых ее применений. А в конкретном приложении могут потребоваться более узкие пределы.

Несколько возможностей ограничения предоставляют свойства полей, которые вы можете увидеть, сделав двойной щелчок на компоненте **Table** и выделив в окне Редактора Полей требуемое поле. Для числовых полей имеются свойства **MinValue** и **MaxValue**, устанавливающие допустимые пределы вводимых в поле значений. Например, если вам требуется принимать на работу сотрудником только в узком возрастном диапазоне, вы можете для поля **Year_b** установить ограничения **MinValue** = 1970 и **MaxValue** = 1980. Эти ограничения не скажутся на отображаемых данных. Но при нарушении этих пределов в процессе редактирования записи или в новой записи будет генерироваться исключение. В результате пользователь увидит сообщение вида, приведенного на рис. 9.29. Вряд ли подобное сообщение на английском языке порадует пользователя. Поэтому лучше перехватить исключение на уровне приложения (см. раздел 12.10.5.2 главы 12) и выдать пользователю понятное пояснение по-русски.

Рис. 9.29.

Сообщение о выходе значения поля за допустимые пределы при использовании свойств **MinValue** и **MaxValue**



Другая возможность ограничения — использование свойств **CustomConstraint** и **ConstraintErrorMessage**. Свойство **CustomConstraint** позволяет написать ограничение на значение поля в виде строки SQL (см. раздел 10.1). Например, для того же поля **Year_b** вы можете написать в свойстве **CustomConstraint**:

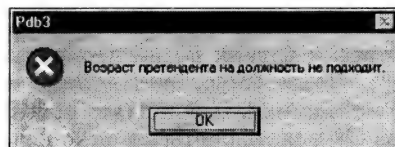
`X < 1980 and X > 1970`

Имя поля (в данном выражении оно обозначено как **X**) может быть произвольным.

Если вы задали свойство **CustomConstraint**, то необходимо задать и свойство **ConstraintErrorMessage**. Оно содержит строку текста, который будет показан пользователю в случае, если он вводит данные, не удовлетворяющие поставленным ограничениям. Например, «Возраст претендента на должность не подходит» (см. рис 9.30, на котором показано окно с этим сообщением). Таким образом, этот вариант обычно много удобнее задания **MinValue** и **MaxValue**, поскольку не требует перехвата исключений. К тому же он может применяться не только к числовым полям.

Рис. 9.30.

Сообщение о выходе значения поля за допустимые пределы при использовании свойств **CustomConstraint** и **ConstraintErrorMessage**



Еще одна возможность проверять данные на уровне поля — использовать обработку события поля **OnValidate**. Это событие возникает перед записью введенного значения поля в буфер текущей записи. Тут можно предусмотреть любые проверки, при появлении недопустимых данных выдать пользователю сообщение и, например, сгенерировать исключение **EAbort** функцией **Abort**. Если все нормально, то после события **OnValidate** возникает еще событие **OnChange**, в обработчике которого тоже еще не поздно сгенерировать исключение.

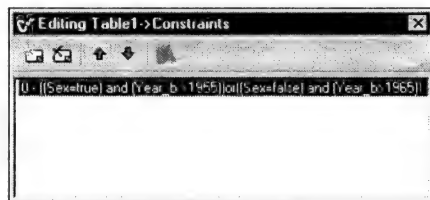
Описанные выше способы проверки данных относятся к конкретному полю. Имеется также возможность осуществлять проверку на уровне записи, анализируя различные ее поля. Для этого служит свойство **Constraints** компонента **Table**. Нажмите кнопку в этом свойстве, и перед вами откроется окно, представленное на рис. 9.31. Щелкая в нем на кнопке **Add New** (крайняя левая) вы можете занести в свойство **Constraints** набор ограничений. Каждое из них представляет собой самостоятельный объект. Выделив одно из ограничений, вы увидите в Инспекторе Объектов его свойства. Свойство **CustomConstraint** представляет собой строку SQL, определяющую допустимые значения. Свойство **ErrorMessage** определяет строку текста, которая будет предъявлена пользователю в случае нарушения ограничений. Например, вы можете написать в свойстве **CustomConstraint**:

```
((Sex=true) and (Year_b >1955))or  
((Sex=false) and (Year_b>1965))
```

а в свойстве **ErrorMessage**: «Принимаем только мужчин >1955 г.р. и женщин >1965 г.р.» Это обеспечит вам различные границы отбора по возрасту для мужчин и женщин.

Рис. 9.31.

Окно редактирования ограничений для записи



Для комплексной проверки данных можно также использовать различные события компонента **Table**, но на этом мы остановимся в последующих разделах при рассмотрении методов программирования работы с данными.

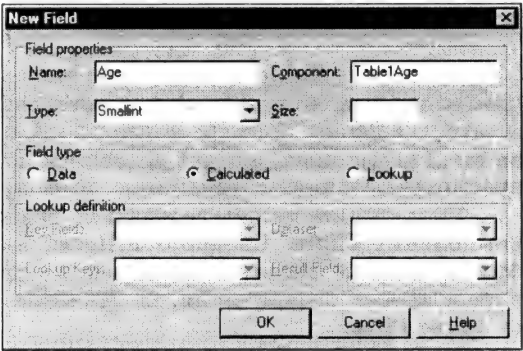
9.5.4 Вычисляемые поля

Теперь попробуем сформировать в таблице новое поле, не предусмотренное при ее создании, значение которого вычисляется на основании значений других полей записи. Подобные поля называются *вычисляемыми полями* (*calculated fields*). Пусть в нашем примере мы хотим добавить поле, вычисляющее возраст сотрудника по его году рождения. Сделайте двойной щелчок на **Table1**, чтобы вызвать Редактор Полей. Щелкните в Редакторе Полей правой кнопкой мыши и во всплывшем меню выберите раздел **New field** (новое поле). Появится окно добавления нового поля, приведенное на рис. 9.32.

В разделе **Field properties** (свойства поля) вы должны указать имя поля (**Name**) — в нашем случае назовем это поле **Age**, тип данных (**Type**) — в нашем случае это **Smallint**, и для некоторых типов — размер (**Size**). Размер указывается для строк и других полей неопределенных размеров.

После ввода всех данных проверьте, переключилась ли группа радиокнопок **Field type** на **Calculated** (это переключение делается автоматически). Затем щелкните на **OK** и вы вернетесь в окно Редактора Полей, причем там появится новое поле **Age**. Задайте для него в Инспекторе Объектов значение **DisplayLabel** равным «Возраст».

Рис. 9.32.
Ввод информации о вычисляемом поле



Вы ввели вычисляемое поле **Age**, но еще не указали программе, как его надо вычислять. Чтобы указать процедуру вычислений, выйдите из Редактора Полей, выделите **Table1**, перейдите в Инспекторе Объектов на страницу событий и щелкните на событии **OnCalcFields**. Это событие наступает каждый раз, как надо обновить значение вычисляемых полей таблицы.

Чтобы вычислить возраст по году рождения, вы можете в обработчике этого события написать оператор:

```
Table1Age->Value = 2000 - Table1Year_b->Value;
```

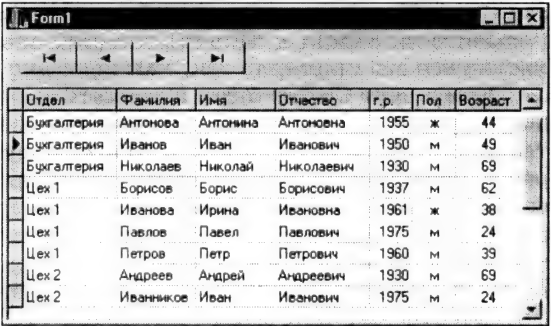
В этом операторе **Table1Age->Value** и **Table1Year_b->Value** — значения полей **Age** и **Year_b** соответственно. Правда, этот оператор рассчитан только на работу в 2000 году, и в дальнейшем его придется ежегодно менять. Можно, добавив в обработчик пару строк, сделать расчет возраста универсальным. Это делает следующий обработчик:

```
void __fastcall TForm1::Table1CalcFields(TDataSet *DataSet)
{
    unsigned short Year, Month, Day;
    Date().DecodeDate(&Year, &Month, &Day);
    Table1Age->Value = Year - Table1Year_b->Value;
}
```

В этом коде введены переменные **Year**, **Month** и **Day** для хранения текущего года, месяца и дня. Использована функция **Date** (см. раздел 15.3.2 в главе 15), возвращающая текущую дату типа **TDateTime** (этот тип используется в C++Builder для хранения дат и времени). И использована функция **DecodeDate** для преобразования этой даты в целые значения года, месяца и дня. В результате переменная **Year** становится равной текущему году (переменные **Month** и **Day** нам не нужны и определены только для того, чтобы можно было обратиться к функции **DecodeDate**).

Запустите приложение и посмотрите, как оно теперь выглядит (рис. 9.33).

Рис. 9.33.
Приложение с введенным вычисляемым полем Возраст



9.5.5 Фильтрация данных

Компонент **Table** позволяет не только отображать, редактировать и упорядочивать данные, но и отфильтровывать записи по определенным критериям. Пользователю в нашем примере могло бы захотеться иметь возможность просматривать не всю базу данных, а отдельно записи по тому или иному отделу, или, например, просмотреть записи сотрудников, имеющих возраст в определенном диапазоне.

Фильтрация может задаваться свойствами **Filter**, **Filtered** и **FilterOptions** компонента **Table**. Свойство **Filtered** включает или выключает использование фильтра. А сам фильтр записывается в свойство **Filter** в виде строки, содержащей определенные ограничения на значения полей. Например, вы можете задать в свойстве **Filter**

```
Dep='Цех 1'
```

установить свойство **Filtered** в **true**, и увидите, что уже в процессе проектирования в таблице отобразятся только те записи, в которых поле **Dep** имеет значение «Цех 1». Обратите внимание на то, что в условии фильтрации строки заключаются в одинарные, а не в двойные кавычки.

В условиях сравнения строк можно использовать символ звездочки «*», который, как в обычных шаблонах, означает: «любое количество любых символов». Например, фильтр

```
Dep='Цех*'
```

приведет к отображению всех записей, в которых значение поля **Dep** начинается с «Цех». В нашем примере будут отображены записи, относящиеся к первому и второму цехам. Но для того, чтобы это сработало, надо, чтобы в опциях, содержащихся в свойстве **FilterOptions** была выключена опция **foNoPartialCompare**, запрещающая частичное совпадение при сравнении (эта опция выключена по умолчанию). Другая опция в свойстве **FilterOptions** — **foCaseInsensitive** делает сравнение строк нечувствительным к регистру, в котором записано условие фильтра. Если включить эту опцию, то слова «Цех 1» и «цех 1» будут считаться идентичными.

При записи условий можно использовать операции отношения **=**, **>**, **>=**, **<**, **<=**, **<>**, а также логические операции **and**, **or** и **not**. Например, вы можете написать фильтр

```
(Dep='Цех 1')and(Year_b<=1970)and(Year_b>=1940)
```

и отобразятся записи сотрудников цеха 1, чей год рождения лежит в заданных пределах. Но использовать в фильтре имена вычисляемых полей (например, введенного нами поля **Age**) не разрешается.

Конечно, свойства, определяющие фильтрацию, можно задавать не только в процессе проектирования, но и программно, во время выполнения. Давайте введем такую возможность в наше приложение. Установите в компоненте **DBGrid1** свойство **Align** равным **alNone**, увеличьте вертикальный размер формы и перенесите на форму группу радиокнопок **RadioGroup** (назовите ее **RGF**), выпадающий список **ComboBox** (назовите его **CBDep**), два элемента **CSpinEdit** со страницы **Samples** (назовите их **SEmin** и **SEmax**), кнопку, написав на ней «Обновить» и метки, разместив все это примерно так, как показано на рис. 9.34.

Компонент **CBDep** будет позволять выбирать подразделение, по которому проводится фильтрация. В его свойство **Items** занесите список имен подразделений в таблице. Компоненты **SEmin** и **SEmax** будут служить для задания диапазона возраста при фильтрации по этому критерию. Задайте в них разумные значения свойств **MaxValue**, **MinValue** и **Value**. В группе радиокнопок **RGF** введите соответствующие надписи (в свойствах **Items** и **Caption**), показанные на рис. 9.34, задайте **ItemIndex** = 0 и **Columns** = 2. Осталось написать операторы, обеспечивающие фильтрацию. Ниже приведен соответствующий текст.

Рис. 9.34.

Приложение с возможностью фильтрации записей

Отдел	Фамилия	Имя	Отчество	г.р.	Пол	Возраст
Бухгалтерия	Иванов	Иван	Иванович	1950	м	49
Цех 2	Сидоров	Сидор	Сидорович	1955	м	44
Бухгалтерия	Антонова	Антонина	Антоновна	1955	ж	44
Цех 1	Петров	Петр	Петрович	1960	м	39
Цех 1	Иванова	Ирина	Ивановна	1961	ж	38
Цех 2	Харитонов	Харитон	Харитонович	1962	м	37
Цех 1	Павлов	Павел	Павлович	1975	м	24

Фильтрация: ☐ Нет ☐ Отдел ☒ Все ☒ Возраст

Отдел: Бухгалтерия Мин: 16 Макс: 50

Обновить

```

unsigned short Year, Month, Day;
//_____
void __fastcall TForm1::Table1CalcFields(TDataSet *DataSet)
{
    Table1Age->Value = Year - Table1Year_b->Value;
}
//_____
void __fastcall TForm1::RGFClick(TObject *Sender)
{
    Table1->IndexName = "depfio";
    if (RGF->ItemIndex == 0)
        Table1->Filtered = false;
    else
    {
        if (RGF->ItemIndex == 2)
            Table1->Filter = "Dep='"+CBDep->Text+"'";
        else if (RGF->ItemIndex == 3)
        {
            Table1->Filter = "(Year_b<="+
                IntToStr(Year-SEmin->Value)+
                ")and(Year_b>="+IntToStr(Year-SEmax->Value)+
                ")";
            Table1->IndexName = "Year";
        }
        else
            Table1->Filter = "(Dep='"+CBDep->Text+
                "')and(Year_b<="+IntToStr(Year-SEmin->Value)+
                ")and(Year_b>="+IntToStr(Year-SEmax->Value)+
                ")";
            Table1->Filtered = true;
    }
}
//_____
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    CBDep->ItemIndex = 0;
    Date().DecodeDate(&Year, &Month, &Day);
    Table1->Active = true;
}
//_____
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
    Table1->Active = false;
}

```

Текст включает три процедуры. Одна из них — **Table1CalcFields** уже была рассмотрена ранее. Ее отличие от рассмотренной заключается только в том, что пе-

ременные **Year**, **Month** и **Day** сделаны глобальными (их объявление вынесено из процедуры) и обращение к функциям **Date** и **DecodeDate** перенесено в процедуру **FormCreate** — обработчик события формы **OnCreate**. Сделано это для того, чтобы можно было воспользоваться значением текущего года **Year** в двух процедурах — **Table1CalcFields** и **RGFClick**. В процедуру **FormCreate** внесен также оператор

```
Table1->Active = true;
```

открывающий связь с базой данных (разрывается эта связь при завершении выполнения приложения в обработчике события формы **OnDestroy**) и оператор

```
CBDep->ItemIndex = 0;
```

Этот оператор задает начальное значение индекса выпадающего списка **CBDep**. Если бы этого оператора не было, то в начале выполнения приложения пользователь не увидел бы текста в списке, поскольку по умолчанию индекс равен -1. Задать же значение индекса в процессе проектирования невозможно, поскольку **ItemIndex** — свойство, доступное только во время выполнения.

Теперь остановимся на основной процедуре приложения — **RGFClick**, которая осуществляет фильтрацию и упорядочивание данных. Обращение к этой процедуре делается при событиях **OnClick** компонента **RGF** — группы радиокнопок, и кнопки **Обновить**. Эта кнопка введена в приложение, чтобы можно было, не переключая радиокнопок, обновлять отображение данных после изменения пользователем имени подразделения или диапазона возраста. Удобно также сделать обращение к процедуре **RGFClick** при событии **OnChange** выпадающего списка отделов.

Первый оператор процедуры **RGFClick** задает индекс **depfio**, обеспечивающий упорядочивание данных по подразделениям и алфавиту. Следующий оператор анализирует индекс группы радиокнопок, т.е. анализирует заданный пользователем способ фильтрации. Если индекс равен нулю (отсутствие фильтрации), то фильтр отключается установкой свойства **Filtered** в **false**. При других значениях индекса это свойство устанавливается в **true**. Если индекс равен 2 (фильтрация по отделам), то значение свойства **Filter** формируется равным **Dep='...'**, где вместо точек фигурирует текст, отображаемый в выпадающем списке **CBDep**. Если индекс равен 3 (фильтрация по возрасту), то значение свойства **Filter** формируется равным **(Year_b<=...and(Year_b>=...))**, где вместо точек подставляются данные, введенные пользователем в компонентах **SEmin** и **SEmax**. При этом приходится пересчитывать заданный пользователем диапазон возраста в диапазон годов рождения, поскольку наложить ограничения непосредственно на вычисляемое поле возраста **Age** невозможно.

Приведенный выше текст рассчитан на применение C++Builder 5. Для более ранних версий C++Builder его надо несколько изменить, поскольку компилятор C++Builder выдаст ошибку вида: «Ambiguity between '_fastcall Sysutils::IntToStr(__int64)' and '_fastcall Sysutils::IntToStr(int)'». Это означает, что компилятор не может осуществить выбор между двумя перегруженными функциями **IntToStr** с аргументами типов **__int64** и **int**. Подробнее об этом см. в разделе 13.2 главы 13. Чтобы ликвидировать эту ошибку, надо применить в функции **IntToStr** явное приведение типа. Например, **IntToStr(int)(Year-SEmin->Value)**.

Вы можете откомпилировать свое приложение и посмотреть его в работе.

Имеется еще один способ проводить фильтрацию — использовать обработку события **OnFilterRecord**. Это событие происходит каждый раз при смене текущей записи, если свойство **Filtered** установлено в **true**. В обработчик события передается по ссылке параметр **Accept** булева типа. Если проверка полей показывает, что запись удовлетворяет фильтру, то **Accept** должен быть установлен в **true**. Если условия фильтрации не выполнены, параметру **Accept** должно быть присвоено значение **false**. Обработчик события **OnFilterRecord**, обеспечивающий те же функции, что и процедура **RGFClick** в приведенном выше примере, может иметь вид

```
void __fastcall TForm1::Table1FilterRecord(TDataSet *DataSet,
                                           bool &Accept)
{
    Accept = (RGF->ItemIndex == 0) ||
        ((RGF->ItemIndex == 2) && (Table1Dep->Value==CBDep->Text)) ||
        ((RGF->ItemIndex == 3) &&
            (Table1Year_b->Value <= (Year-SEmin->Value)) &&
            (Table1Year_b->Value >= (Year-SEmax->Value))) ||
        ((Table1Dep->Value == CBDep->Text) &&
            (Table1Year_b->Value <= (Year-SEmin->Value)) &&
            (Table1Year_b->Value >= (Year-SEmax->Value)));
}
```

В этом операторе свойству **Accept** непосредственно присваивается значение результата анализа условий фильтрации в зависимости от значения свойства **ItemIndex** группы радиокнопок **RGF**. Например, если **RGF->ItemIndex = 0** (пользователь задал просмотр без фильтрации), то **Accept = true** независимо от каких-либо других условий.

Приведенный оператор обеспечивает нужную фильтрацию. Но чтобы он работал, необходимо выполнить еще два условия. Во-первых, надо не забыть установить в компоненте **Table1** значение **Filtered = true**. Кроме того, надо обеспечить, чтобы при смене условий отображения таблицы проводилась бы новая фильтрация. Иначе просто события **OnFilterRecord** не будут происходить. Поэтому в обработчик **RGFClick** события **OnClick** компонента **RGF** надо вставить операторы:

```
Table1->Filtered = false;
Table1->Filtered = true;
if (RGF->ItemIndex == 3)
    Table1->IndexName = "Year";
else Table1->IndexName = "depfio";
```

Первые два оператора обеспечивают отключение и включение фильтрации. При включении и происходят события **OnFilterRecord**. А третий оператор просто изменяет индексацию в зависимости от того, что задал пользователь. На этот обработчик **RGFClick** надо сослаться и в событии **OnClick** кнопки Обновить.

Попробуйте создать описанный вариант приложения, откомпилировать его и посмотреть в работе.

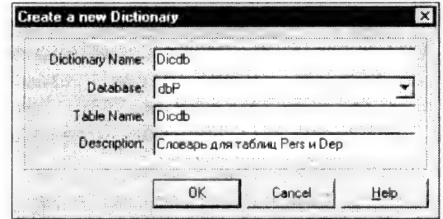
9.6 Использование словарей атрибутов полей

Выше мы рассмотрели множество свойств полей. Если вы хотите использовать эти свойства в различных приложениях или в различных частях одного приложения, то желательно обеспечить единый стиль представления данных, единые заголовки одноименных столбцов таблиц, их единый размер и т.п. Задавать для каждого нового компонента **Table** все это заново — достаточно трудоемкое занятие. К тому же обеспечить при этом единообразие довольно сложно.

Решить эту задачу помогают словари. Создать новый словарь можно с помощью программы **SQL Explorer**, вызываемой командой главного меню **Database | Explore**. Ранее в разделе 9.3.4 мы рассмотрели страницу **Databases SQL Explorer**. Для создания нового словаря надо перейти на страницу **Dictionary** и выполнить команду **Dictionary | New**. Откроется диалоговое окно создания нового словаря, представленное на рис. 9.35. В верхнем окне редактирования **Dictionary Name** вы пишете имя создаваемого словаря. Это то имя, по которому вы в дальнейшем сможете выбрать этот словарь среди других. В расположенном ниже выпадающем списке **Database** выбираете базу данных, в которой хотите сохранить словарь. Словарь по умолчанию сохраняется в виде таблицы **Paradox**. Поэтому в следующем окне **Table Name** вы пишете имя таблицы, в которой будет храниться словарь. В нижнем окне

Рис. 9.35.

Создание нового словаря в SQL Explorer

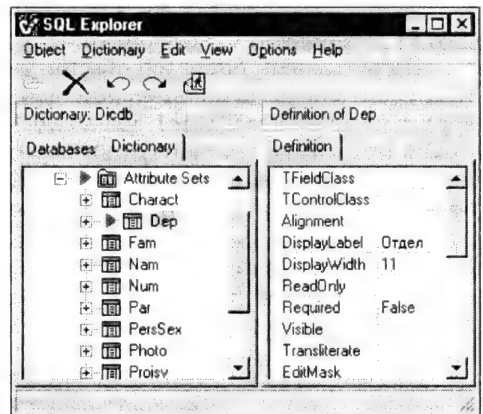


Description вы можете написать произвольный комментарий, описывающий назначение вашего словаря. В заключение щелкаете на OK и новый словарь создан.

Вы попадаете в окно, представленное на рис. 9.36. Раскрыв вершину Dictionary, вы увидите два раздела: Databases — базы данных, и Attribute Sets — множества атрибутов. Пока обе эти вершины будут пустыми.

Рис. 9.36.

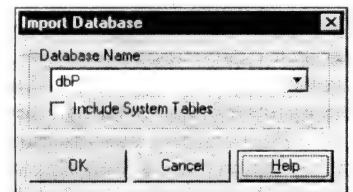
Задание атрибутов полей в словаре



Чтобы связать создаваемый словарь с таблицами базы данных, надо выполнить команду Dictionary | Import From Database. В появившемся диалоговом окне (рис. 9.37) вы можете выбрать из выпадающего списка псевдоним базы данных. После того, как вы нажмете OK, сведения о базе данных, ее таблицах и полях таблиц будут импортированы в словарь. В окне рис. 9.36 в вершине Databases появятся дочерние вершины относящиеся к базе данных, таблицам и полям. Можно создавать словарь и на основе нескольких баз данных, поочередно импортируя их командой Dictionary | Import From Database в словарь.

Рис. 9.37.

Импорт базы данных в словарь



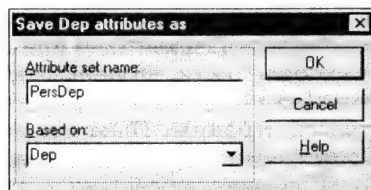
Теперь можно задавать атрибуты полей. Вы можете создавать новые дочерние вершины множества атрибутов непосредственно в этом окне. Для этого надо выделить раздел Attribute Sels, щелкнуть правой кнопкой мыши и из всплывающего меню выбрать раздел New. Затем в появившейся на экране дочерней вершине надо ввести имя поля. После этого в правой части окна можно задавать различные атрибуты.

Но, конечно, много проще перенести атрибуты из какого-то из ваших проектов, где они уже установлены в Редакторе Полей. Откройте какой-нибудь из ва-

ших предыдущих проектов, в котором вы установили атрибуты всех полей таблицы Pers. Сделайте в нем двойной щелчок на компоненте **Table**. Если вы пользовались при создании этого приложения Редактором Полей, то в его окне появится список полей. Для записи атрибутов этих полей в словарь выделите их все, щелкните правой кнопкой мыши и из всплывшего меню выберите раздел **Save Attributes as**. Вам откроется небольшое диалоговое окно, представленное на рис. 9.38. В его верхнем окошке **Attribute set name** вам предлагается имя сохраняемого множества атрибутов, которое по умолчанию складывается из имени таблицы и имени поля. Вы можете изменить его, как угодно. Например, убрать из него имя таблицы. В нижнем окошке **Based on** вы можете выбрать одно из уже имеющихся в словаре полей, в качестве основы для атрибутов данного поля. Например, после того, как вы введете в словарь все атрибуты полей таблицы Pers, вы можете аналогичным образом вводить атрибуты полей таблицы Dep. При этом для атрибутов поля Dep вы можете выбрать за основу уже введенные атрибуты **PersDep**.

Рис. 9.38.

Сохранение атрибутов полей из Редактора Полей в словарь



После того, как вы щелкнете в окне рис. 9.38 на **OK**, в словаре появится данное множество атрибутов с указанным вами именем. Эта процедура повторится для каждого из выделенных вами полей, атрибуты которых вы сохраняете.

Теперь повторите опять команду главного меню **Database | Explore**. Вы увидите окно, показанное ранее на рис. 9.36, но в нем еще не будет введенных вами вершин. Для того, чтобы увидеть их, щелкните правой кнопкой мыши и выберите из всплывшего меню команду **Refresh** — обновить. Тогда изображение обновится и вы увидите вершины, соответствующие сохраненным атрибутам полей. В правой половине окна отображаются атрибуты, которые при желании вы можете изменять. Причем вы можете увидеть, что сохраняются все атрибуты объектов-полей, включая заданные ограничения, значения по умолчанию, атрибуты оформления и т.д.

Прежде, чем расстаться с окном **SQL Explorer**, несколько слов о методике редактирования в нем. Вы можете изменять атрибуты, можете удалять вершины (команда **Object | Delete** или вторая слева быстрая кнопка). И то, и другое не выполняется сразу. Просто изменяемая вершина помечается на удаление или редактирование. От изменения вершины можно отказаться (команда **Object | Cancel** или третья слева быстрая кнопка). А можно подтвердить изменение командой **Object | Apply** или четвертая слева быстрая кнопка.

Команда **Dictionary | Select** позволяет открыть один из зарегистрированных словарей. Если вы хотите в разрабатываемом вами новом приложении воспользоваться определенным словарем — откройте его данной командой (если, конечно, он не был открыт ранее).

А теперь посмотрим, как можно использовать словарь. Откройте новое приложение, перенесите на форму обычную цепочку компонентов **Table**, **DataSource**, **DBGrid**, свяжите компоненты друг с другом и с таблицей Pers базы данных **dbP**. Установите в **Table** свойство **Active** в **true**. Вы увидите содержание таблицы, но оно пока не будет отформатировано так, как нужно вам. А теперь сделайте двойной щелчок на **Table** и добавьте в Редактор Полей все поля таблицы. В тот же момент вы увидите, что восприняты все форматы отображения полей и все их атрибуты, которые вы сохраняли в словаре. Чувствуете, сколько своего времени вы сэкономили? При желании вы можете отсоединить то или иное поле от словаря, выделив его в Редакторе Полей, щелкнув правой кнопкой и выбрав команду **Unassolate**

attributes. После этого вы можете или изменить атрибуты, или взять за образец атрибуты какого-либо другого множества из словаря. Для этого щелкните правой кнопкой и выберите команду *Associate attributes*. Вам будет предложено окно со списком всех множеств атрибутов в словаре. Из этого списка вы можете выбрать образец для своего поля, а затем отредактировать его.

9.7 Некоторые компоненты визуализации и управления данными

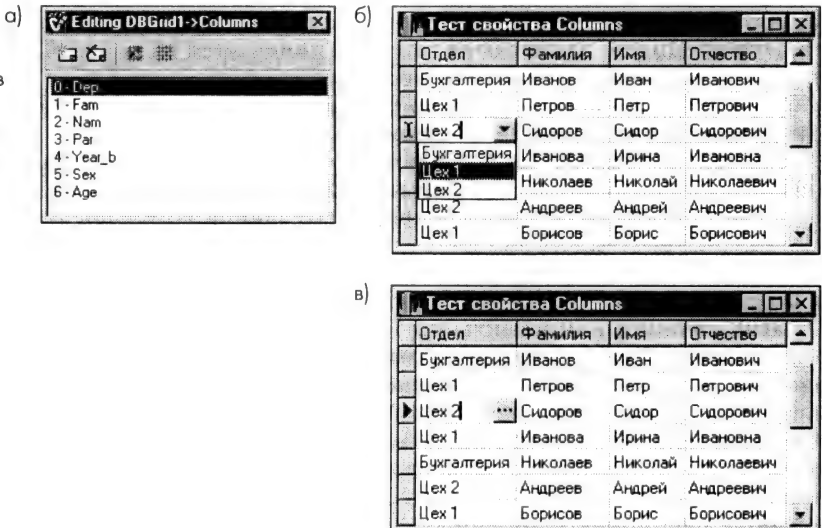
В приведенном приложении в качестве компонента визуализации и управления данными использовался **DBGrid**. Но еще не со всеми свойствами этого компонента мы ознакомились. В том виде, как мы его применяли, пользователю неудобно вводить многие данные. Например, вводя имя подразделения, пользователь должен полностью его написать: «Бухгалтерия» или «Цех 1». Да еще, если он ошибется в названии, то приложение выдаст ему ошибку. Этот недостаток можно устранить, если воспользоваться свойством **Columns** компонента **DBGrid**. Это свойство представляет собой набор объектов, каждый из которых отражает один столбец таблицы. Пока мы использовали столбцы, формируемые по умолчанию. Попробуем теперь сформировать их, используя свойство **Columns**.

Щелкните правее этого свойства в Инспекторе Объектов. На экране появится пустое окно Редактора Столбцов (рис. 9.39 а). Вы можете добавлять столбцы по одному, щелкая на быстрой кнопке **Add** (первая слева) и указывая для них в Инспекторе Объектов соответствующие поля в свойстве **FieldName**. То же самое можно делать, щелкая правой кнопкой мыши и выбирая в контекстном меню раздел **Add**. А можно поступить иначе: выбрать из контекстного меню раздел **Add All Fields** или щелкнуть на соответствующей быстрой кнопке (вторая справа). В окне Редактора Столбцов появится список столбцов всех полей, которые добавлены вами в Редактор Полей компонента **Table**. Затем кнопкой **Delete** (вторая слева) вы можете удалить столбцы, ненужные в таблице.

Выделите в Редакторе Полей столбец, связанный с полем **Dep**, и посмотрите его свойства в Инспекторе Объектов. Среди многих достаточно очевидных свойств столбца вы увидите свойство **ButtonStyle**. Оно определяет стиль ввода данных в поле текущей записи. Свойство **ButtonStyle** может принимать значения:

Рис. 9.39.

Главное окно редактора столбцов (а) и ячейка поля **Dep** при **ButtonStyle = cbsAuto** (б) и при **ButtonStyle = cbsEllipsis** (в)



cbsAuto	Появление при редактировании кнопки, связанной с выпадающим списком допустимых значений
cbsEllipsis	Появление при редактировании кнопки с многоточием «...», при щелчке на которой наступает событие OnEditButtonClick компонента DBGrid
cbsNone	Обычное редактирование без каких-либо кнопок

Если установить значение **cbsAuto** (оно принято по умолчанию), то можно дополнительно установить свойство столбца **PickList**, которое представляет собой список допустимых значений поля. В нашем примере для столбца **Dep** можно занести в список **PickList** значения «Бухгалтерия», «Цех 1» и «Цех 2». Тогда при попытке пользователя редактировать данное поле в соответствующей ячейке таблицы появится кнопка, связанная с выпадающим списком, содержащим допустимые значения (рис. 9.39 б). Аналогично можно было бы задать значения «м» и «ж» в списке **PickList** поля **Sex**. Свойство **DropDownRows** (по умолчанию 7) определяет допустимое число строк в списке, отображаемое без появления полосы прокрутки. Если реальное число строк меньше **DropDownRows**, размер списка устанавливается автоматически.

Аналогично происходит редактирование, если значение **cbsAuto** относится к столбцу, содержащему поле просмотра (см. раздел 9.10.2). Тогда выпадающий список заполняется автоматически. Если же данное поле не является результирующим полем просмотра и список **PickList** не заполнен, то никакой кнопки при редактировании не появится.

Если значение свойства **ButtonStyle** равно **cbsEllipsis**, то при попытке пользователя редактировать данное поле в нем появляется кнопка с многоточием «...» (см. рис. 9.39 в). При щелчке на этой кнопке наступает событие **OnEditButtonClick** компонента **DBGrid**. В обработчике этого события вы можете предусмотреть показ пользователю какого-то списка возможных значений или предложить диалог, в котором пользователь может в удобной форме выбрать требуемое значение. В обработчике события **OnEditButtonClick** вы можете узнать, какое поле редактируется, по свойству **SelectedField** компонента **DBGrid**. Это же свойство позволит вам занести в поле установленное пользователем значение. Например:

```
if(DBGrid1->SelectedField == Table1Dep)
{
    ...
    DBGrid1->SelectedField->Value = ...;
}
```

Если значение свойства **ButtonStyle** равно **cbsNone**, то редактирование поля производится без каких-либо подсказок в виде кнопок.

Компонент **DBGrid** — удобный, но далеко не единственный и не всегда лучший компонент визуализации. Форма представления данных в нем не всегда удобная: слишком регулярная, без разделителей, не акцентирующая внимания пользователя на каких-то группах полей. Имеется много других компонентов визуализации и управления данными, расположенных на странице **Data Control** библиотеки компонентов, которые нередко предпочтительнее **DBGrid**.

Отметим некоторые из этих компонентов.

DBText — аналог обычной метки **Label**, но связанный с данными. Он позволяет отображать данные некоторого поля, но не дает возможности его редактировать. Тип отображаемого поля может быть различным: строка, число, булева величина. Компонент автоматически переводит соответствующие типы в отображаемые символы.

DBEdit — связанный с данными аналог обычного окна редактирования **Edit**. Он позволяет отображать и редактировать данные полей различных типов: строка, число, булева величина. Впрочем, если задать в компоненте **ReadOnly=true**, то он, как и **DBText**, будет служить элементом отображения, но несколько более изящным, чем **DBText**.

DBMemo — связанный с данными аналог обычного многострочного окна редактирования **Memo**. Он позволяет отображать и редактировать данные поля типа **Memo**, а также данные любых типов, указанных выше для предыдущих компонентов. Например, в этом компоненте можно отображать и редактировать характеристику сотрудника, если такое поле предусмотрено в таблице.

DBRichEdit — связанный с данными аналог обычного многострочного окна редактирования текста в обогащенном формате **RTF**. Область применения та же, что для компонента **DBMemo**.

DBImage — связанный с данными аналог обычного компонента **Image**. Компонент позволяет отображать графические поля, например, фотографии сотрудников.

DBCheckBox — связанный с данными аналог обычного индикатора **CheckBox**. Он позволяет отображать и редактировать данные поля булева типа. Если при выводе данных поле имеет значение **true**, то индикатор включается. И наоборот, при редактировании поля присваиваемое ему значение определяется состоянием индикатора. Кстати, это еще один способ обеспечить пользователю безошибочный ввод значений в булево поле, помимо рассмотренного выше задания списка возможных значений в **DBGrid**.

DBRadioGroup — связанный с данными аналог группы радиокнопок **RadioGroup**. Компонент позволяет отображать и редактировать поля с ограниченным множеством возможных значений. В нашем примере это может относиться к полю **Dep** или к полю **Sex**. Свойство **Items**, как и в обычной группе радиокнопок, определяет число кнопок и надписи около них. Но есть еще свойство **Values**, которое определяет значения полей, соответствующие кнопкам. Таким образом, надписи у кнопок и значения полей в общем случае могут не совпадать друг с другом. Если свойство **Values** не задано, то в качестве значений полей берутся строки из свойства **Items**, т.е. надписи около кнопок. Компонент **DBRadioGroup** предоставляет еще одну возможность ввода пользователем данных путем выбора, а не написанием полного значения поля.

Есть еще несколько компонентов визуализации и управления данными, которые мы рассмотрим несколько позднее. Все перечисленные компоненты имеют два основных свойства: **DataSource** — источник данных (компонент типа **TDataSource**) и **DataField** — поле, с которым связан компонент.

Характерной особенностью всех этих компонентов, отличающей их от аналогичных компонентов, не связанных с данными, является отсутствие в окне Инспектора Объектов их основных свойств, отображающих содержание: **Caption** в **DBText**, **Text** в **DBEdit**, **Picture** в **DBImage** и т.п. Все эти свойства имеются в компонентах, но они доступны только во время выполнения. Так что программно их можно читать, редактировать, устанавливать, но во время проектирования задать их значения невозможно. Это естественно, так как эти свойства — это значения соответствующих полей таблицы базы данных.

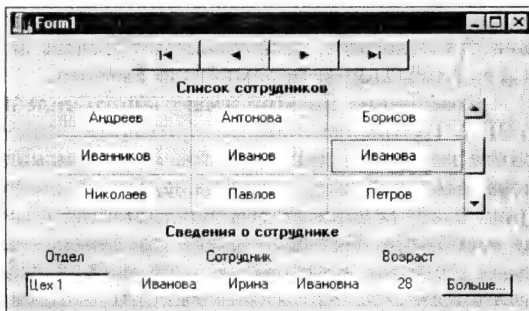
Перечисленные компоненты, как, впрочем, и любые обычные метки и другие компоненты, могут размещаться непосредственно на форме, на любой панели, а могут размещаться в специальном компоненте — таблице **DBCtrlGrid**. Этот компонент состоит из нескольких панелей, из которых проектируется только первая, а остальные повторяют структуру первой, но относятся к последующим записям таблицы. **DBCtrlGrid** имеет свойство **DataSource**, которое автоматически передается всем расположенным на панелях компонентам, связанным с данными. Свойство **RowCount** определяет число строк в таблице (по умолчанию 3), а свойство

ColCount — число колонок в таблице (по умолчанию 1). Ширину и высоту панелей **DBCtrlGrid** можно задавать свойствами **PanelWidth** и **PanelHeight** соответственно, или задавая общую ширину и высоту компонента свойствами **Width** и **Height**. В этом случае C++Builder может автоматически округлить заданные вами значения так, чтобы обеспечить равные ширину и высоту всех панелей.

Опробуйте эти компоненты в работе. На рис. 9.40 показано приложение, построенное с использованием рассмотренных элементов. Компонент **DBCtrlGrid** имеет 3 строки и 3 столбца. На панели этого компонента в данном случае расположен всего один компонент **DBText**, связанный с полем **Fam**. В целом весь компонент **DBCtrlGrid** отображает алфавитный список сотрудников (таблица использует индекс **fio**). В нижней части экрана располагается подробная информация о сотруднике. Отдел, в котором работает сотрудник, отображается компонентом **DBEdit**, а остальные — компонентами **DBText**. Компоненты, отображающие фамилию, имя и отчество, расположены без промежутков, так что образуют сплошную полосу. В компоненте **DBEdit** установлено **ReadOnly=false**, так что пользователь может редактировать эту информацию. Таким образом, перемещаясь по списку сотрудников, пользователь может просматривать информацию о выбранном сотруднике и частично ее редактировать, перемещая, например, с помощью окна **DBEdit** сотрудника в другое подразделение.

Рис. 9.40.

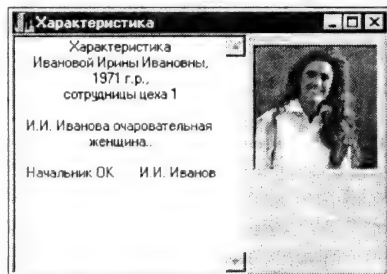
Приложение, демонстрирующее использование различных компонентов ввода и отображения данных



Кнопка **Больше** в правом нижнем углу позволяет открыть дополнительное окно (рис. 9.41), в котором отображается характеристика и фотография сотрудника. Для этого используются соответственно компоненты **DBImage** и **DBMemo**. Характеристику, загружаемую в **DBMemo**, пользователь может редактировать.

Рис. 9.41.

Вспомогательное окно приложения, отображающее характеристику и фотографию сотрудника



Размещение компонентов визуализации на отдельной форме не вызывает никаких принципиальных сложностей. Надо только в модуле дополнительной формы сослаться с помощью директивы препроцессора **#include** на модуль основной формы, чтобы получить доступ к расположенному на ней компоненту типа **TDataSource**. Вспомним, что эту ссылку проще всего сделать командой **File | Include Unit Hdr**. Тогда при задании свойств **DataSource** компонентов, расположенных на вспомогательной форме, в выпадающем списке появится источник данных **Form1->**

DataSource1, который и надо выбрать. А в главной форме надо аналогичным образом сослаться на вспомогательную форму, чтобы в нужный момент можно было сделать ее видимой. Тогда в обработчике события **OnClick** кнопки Больше можно написать оператор:

```
if (! Form2->Visible) Form2->Show();
```

который делает вспомогательную форму **Form2** видимой, если она не была видимой до этого.

В рассмотренном приложении имеются определенные неудобства: результаты редактирования будут заноситься в базу данных только тогда, когда пользователь перейдет к следующей записи. Кроме того, при неверном указании отдела пользователь увидит сообщение об ошибке на английском языке, которое, возможно, будет ему мало понятно. Все это не трудно было бы исправить. Позднее мы еще вернемся к этому приложению и посмотрим, как его можно усовершенствовать.

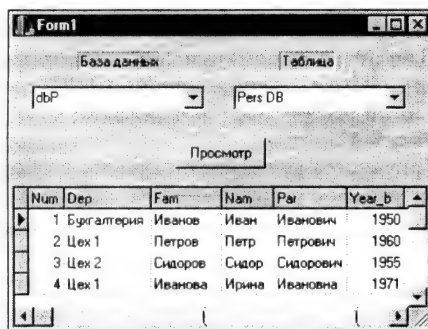
9.8 Компонент Session

Компоненты **Session** осуществляют общее управление связыванием приложения с базами данных. Обычно пользователю не приходится заботиться о компоненте **Session**, поскольку **C++Builder** автоматически генерирует объект **Session** в каждом приложении, работающем с базами данных. На этот объект можно ссылаться через глобальную переменную **Session**.

Компонент **Session** имеет много полезных методов и позволяет легко работать с BDE. Приведем простой пример использования некоторых методов **Session**. Давайте построим приложение, позволяющее просматривать любую заданную пользователем таблицу в любой заданной им доступной базе данных. Это приложение (рис. 9.42) содержит два выпадающих списка типа **TComboBox**, названных **cbAlias** и **cbTable**. Первый из них предназначен для выбора пользователем псевдонима базы данных, а второй — для выбора таблицы. Кнопка Просмотр предназначена для просмотра выбранной таблицы в компоненте типа **TDBGrid**, связанном цепочкой ссылок с компонентами **DataSource1** и **Table1**.

Рис. 9.42.

Приложение, позволяющее пользователю выбрать любую доступную базу данных и любую таблицу в ней



Чтобы такое приложение функционировало, надо при создании формы загрузить список **cbDatabase** доступными BDE псевдонимами. Это делается следующим оператором, помещенным в обработчик события **OnCreate** формы:

```
Session->GetAliasNames(cbAlias->Items);
```

Этот оператор использует метод **GetAliasNames** объекта **Session**, который передает в свой параметр типа **TStrings** (в данном примере это список **cbAlias->Items**) перечень псевдонимов баз данных, зарегистрированных в BDE.

При выборе пользователем в списке **cbAlias** базы данных надо загрузить список **cbTable** перечнем таблиц выбранной базы данных. Это делается включением в обработчик события **OnChange** компонента **cbAlias** операторов:

```
Session->GetTableNames(cbAlias->Text, "", true, false, cbTable->Items);  
cbTable->ItemIndex = 0;
```

Первый из этих операторов использует метод **GetTableNames** для загрузки в свой последний параметр типа **TStrings** (в данном примере это список **cbTable->Items**) перечня таблиц базы данных, заданной своим первым параметром (в данном примере это **cbAlias->Text**). Второй параметр метода позволяет задать шаблон, отбирающий имена таблиц. Например, шаблон «р*» отберет имена таблиц, начинающиеся с символа «р». Пустой шаблон, использованный в примере, означает выбор всех таблиц. Третий параметр, установленный в **true**, означает, что в имена таблиц будет включаться расширение файла (это необходимо для таблиц **Paradox** и **dBase**). Четвертый параметр задается равным **false** для баз данных **Paradox** и **dBase**, а для баз данных, основанных на **SQL**, этот параметр устанавливается в **true**, чтобы возвращать и таблицы данных и системные таблицы, определяющие структуру данных.

Теперь остается ввести в приложение обработчик нажатия кнопки **Просмотр**. Этот обработчик может иметь вид:

```
if (cbTable->Text == "")  
{  
    ShowMessage("Не задана таблица");  
    return;  
}  
Table1->Active = false;  
Table1->DatabaseName = cbAlias->Text;  
Table1->TableName = cbTable->Text;  
Table1->Active = true;
```

Этот обработчик выводит компонент **Table1** из активного состояния, чтобы можно было связаться с новой базой данных, затем задает его свойства **DatabaseName** и **TableName** равными выбранным пользователем значениям и переводит **Table1** в активное состояние.

Откомпилируйте приложение и посмотрите его в работе. Вы можете посмотреть с его помощью не только свои, но и другие базы данных, прилагаемые к **C++Builder**.

Выше показано использование компонента **Session**, создаваемого неявно по умолчанию. Вводить явным образом компоненты **Session** приходится только в многозадачных приложениях, в которых предусмотрено несколько параллельных процессов со своими потоками обмена информацией с базой данных. В таких многопоточных приложениях обычно вводится явным образом по одному компоненту **Session** на каждый поток, чтобы исключить влияние потоков друг на друга.

При явном вводе компонента **Session** в приложение следует установить его свойство **SessionName** — имя сеанса сетевого соединения, задав в нем произвольный идентификатор, например, **s1**. После этого в выпадающих списках свойств **SessionName** компонентов типа **TDatabase**, **TTable**, **TQuery** и т.п. появится введенное вами имя. Выбор соответствующего имени сеанса сетевого соединения из этих списков свяжет эти компоненты с соответствующим компонентом **Session**.

Если в приложении имеется несколько компонентов **Session**, глобальная переменная **Session** по-прежнему одна. Она имеет тип **TSessionsList** и содержит данные о каждом компоненте **Session**.

9.9 Компонент BatchMove

Компонент **BatchMove** предназначен для групповых операций переноса данных из одного набора в другой. Основные свойства компонента: **Source** — набор источника данных (например, компонент типа **TTable**) и **Destination** — приемник данных типа **TTable**. Свойство **Mode** определяет режим переноса данных. Это свойство может иметь следующие значения:

batAppend	Записи из источника данных добавляются в приемник, не изменяя существующих там записей. Таблица-приемник должна существовать до начала переноса данных
batUpdate	Записи в таблице-приемнике с ключевыми полями, соответствующими полям в таблице-источнике, изменяются на записи из источника. Новые записи в приемник не добавляются. Таблица-приемник должна существовать до начала переноса данных и должна иметь индекс
batAppendUpdate	Записи в таблице-приемнике с ключевыми полями, соответствующими полям в таблице-источнике, изменяются на записи из источника. Записи из таблицы источника, которые не имеют соответствия в приемнике, добавляются туда. Таблица-приемник должна существовать до начала переноса данных и должна иметь индекс
batDelete	Записи в таблице-приемнике, которым находится соответствие в источнике, удаляются из приемника. Таблица-приемник должна существовать до начала переноса данных и должна иметь индекс
batCopy	Таблица-приемник создается и заполняется записями источника. Если таблица-приемник уже существовала, то ее содержимое заменяется на содержимое источника

Основной метод компонента — **Execute** выполняет операцию переноса данных. Свойство **MovedCount** указывает число записей, успешно перенесенных в таблицу-приемник.

При переносе записей из одной таблицы в другую могут возникать проблемы, связанные с несоответствием типов полей в таблице-источнике и таблице-приемнике. В этих случаях свойство **AbortOnProblem** указывает, должна ли немедленно прекращаться операция, вызвавшая несоответствие типов полей в таблице-источнике и таблице-приемнике. При задании **AbortOnProblem = false** желательно одновременно задать свойство **ProblemTableName**. Это свойство указывает имя таблицы **Paradox**, в которую будут помещаться записи, в которых обнаружено несоответствие типов полей и которые поэтому не помещены в приемник. Свойство **ProblemCount** определяет число записей, в которых возникли подобные проблемы. При задании **AbortOnProblem = false** записи с полями несоответствующих типов нередко все-таки могут помещаться в таблицу-приемник. Дело в том, что компонент **BatchMove** пытается в этом случае преобразовать тип поля таблицы-источника в тип поля таблицы-приемника. Если такое преобразование возможно, то никаких проблем при переносе записей не возникает.

В некоторых случаях перенос отдельных записей может оказаться невозможным из-за того, что они нарушают целостность данных в таблице-приемнике, например, дублируют значения ключевого поля, которые должны быть уникальными. В этих случаях свойство **AbortOnKeyViol** указывает, должна ли немедленно прекращаться операция, вызвавшая нарушение целостности данных в табли-

це-приемнике. При задании **AbortOnKeyViol** = **false** желательно одновременно задать свойство **KeyViolTableName**. Это свойство указывает имя таблицы **Paradox**, в которую будут помещаться записи, которые не перенесены в приемник из-за нарушения целостности данных. Свойство **KeyViolCount** определяет число таких записей.

Поскольку при переносе данных могут возникать указанные выше и иные неприятности, в компоненте **TBatchMove** предусмотрен механизм, позволяющий произвести «откат» и сохранить прежние данные таблицы-приемника. Этому служит свойство **ChangedTableName**, в котором вы можете указать имя таблицы **Paradox**, создаваемой для сохранения копий всех изменяемых записей таблицы-приемника. Свойство **ChangedCount** содержит число записей, измененных или добавленных в таблице-приемнике (при **Mode** = **batUpdate** или **batAppendUpdate**), или удаленных из нее (при **Mode** = **batDelete**). Копии этих записей хранятся в таблице **ChangedTableName**.

Свойство **Mappings** типа **TStrings** позволяет задать таблицу соответствия полей источника и приемника. По умолчанию поля таблиц переносятся в поля приемника с теми же именами и в той же последовательности, как в источнике. Если это вас устраивает, то свойство **Mappings** задавать не надо. Но если вы зададите свойство **Mappings**, то можете изменить определенный по умолчанию способ переноса данных. Во-первых, вы можете задать в этом свойстве только часть полей, значения которых надо переносить. Имена полей записываются по одному в строке. Значения полей, не указанные в **Mappings**, переноситься не будут. Более того, вы можете указать, в какое поле приемника надо переносить значение поля источника. Например, если вы задаете в **Mappings** строку

```
Dep
```

то значение поля **Dep** из источника будет переноситься в поле **Dep** приемника. Но если вы зададите строку

```
DepNew = Dep
```

то значение поля **Dep** из источника будет переноситься в поле **DepNew** приемника.

9.10 Приложения с несколькими связанными таблицами

9.10.1 Связь головной и вспомогательной таблиц

Предыдущие приложения общались с одной таблицей. Теперь посмотрим, как строить приложения с несколькими связанными друг с другом таблицами.

Две таблицы могут быть связаны друг с другом по ключу. Одна из этих связанных таблиц является головной (*master*), а другая — вспомогательной, детализирующей (*detail*). Например, мы хотим построить приложение, в котором имеется таблица **Dep**, содержащая список подразделений учреждения (поле **Dep**) и характеристику этих подразделений (поле **Proizv** булева типа, в котором **true** означает «Производство», а **false** — «Управление»). И хотим, чтобы пользователь, перемещаясь по этой таблице, видел не только характеристику подразделения, но и список сотрудников этого подразделения, т.е. записи из таблицы **Pers**, в которых значение поля **Dep** совпадает со значением поля **Dep** в текущей записи первой таблицы.

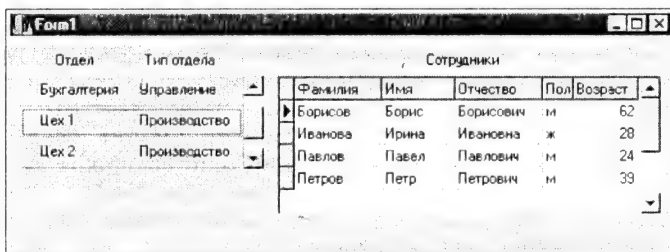
В этом случае головной является таблица **Dep**, вспомогательной — таблица **Pers**, а ключом, определяющим их связь, являются поля **Dep** из обеих таблиц.

Откройте новое приложение и разместите на нем 2 комплекта **Table**, **Data-Source** и средств отображения данных. Результат может выглядеть например так, как показано на рис. 9.43. В первом комплекте использован компонент отображения **DBCtrlGrid**, на панелях которого располагаются метки **DBText**. Комплект

обычным образом настраивается на первую, головную (master) таблицу — Dep. Таблица должна быть индексируема по полю Dep, которое будет ключом для связи таблиц. Метки, размещенные на DBCtrlGrid, связываются с полями Dep и Proizv.

Рис. 9.43.

Простое приложение, работающее с двумя таблицами данных



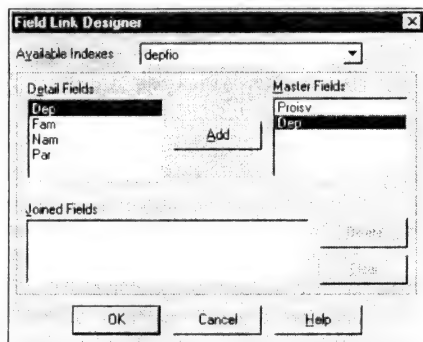
Второй комплект компонентов использует для отображения данных компонент **DBGrid**. Комплект настраивается на вторую вспомогательную таблицу — Pers. Для таблицы должен быть задан индекс, содержащий ключевое поле связи Dep. После того, как все это сделано, можете выполнить приложение, чтобы убедиться, что все сделано правильно. Пока вы имеете две несвязанные друг с другом таблицы, по которым можете перемещаться независимо.

Теперь свяжите эти таблицы. Разорвите временно связь с базой данных во втором комплекте, настроенном на вспомогательную таблицу Pers (установите **Active = false**). Далее в свойстве **MasterSource** компонента **Table**, настроенного на вспомогательную таблицу, установите имя головной таблицы. После этого щелкните на свойстве **MasterFields**. Откроется окно редактора связей полей (Field Link Designer). Его вид приведен на рис. 9.44. В нем слева в окне Detail Fields расположены имена полей вспомогательной таблицы, но только тех, по которым таблица индексируется. Именно поэтому надо индексировать таблицу так, чтобы индекс включал ключевое поле связи Dep. Справа в окне Master Fields расположены поля головной таблицы. Выделите в одном и другом окне ключевое поле (в нашем случае Dep). При этом активизируется кнопка Add, и после щелчка на ней ключевые поля переносятся в нижнее окно Joined Fields — соединяемые поля. Если ключ составной (например, фамилия, имя, отчество) — эта операция повторяется для других полей. В конце формирования связей щелкните на OK и в свойстве **MasterFields** компонента **Table** появится текст — связанные поля. После этого можете восстановить связь с базой данных (**Active = true**) и запустить приложение.

Вы увидите (рис. 9.43), что в зависимости от того, какую запись вы выделяете в списке отделов, вам отображается список сотрудников этого отдела. Таким образом, курсор скользит по головной таблице, а вспомогательная таблица отображает только те записи, в которых ключевые поля совпадают с ключевыми полями головной таблицы.

Рис. 9.44.

Окно редактора связей полей головной и вспомогательной таблиц

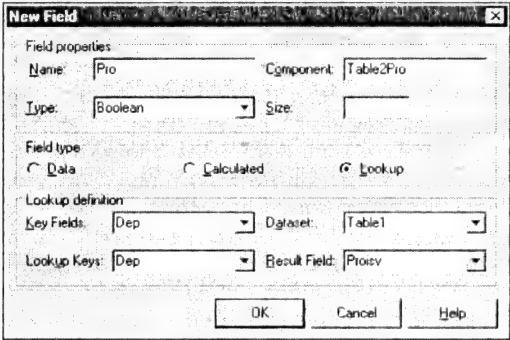


9.10.2 Поля просмотра (lookup fields)

Возможна и другая связь таблиц, отличная от рассмотренной выше. Вы можете ввести в компонент **Table**, связанный с одной таблицей, поля просмотра, значения которых получаются в результате просмотра другой таблицы. Эти поля содержат данные только для чтения. Пусть, например, вы хотите добавить в компонент **Table2**, связанный с таблицей **Pers**, поле, которое бы отображало характер подразделения, в котором работает каждый сотрудник. Этот характер подразделения содержится в таблице **Dep** в поле **Proizv**. Пусть с этой таблицей связан компонент **Table1**.

Сделайте двойной щелчок на **Table2** — таблице, в которую вы хотите ввести поле просмотра. В появившемся окне Редактора Полей щелкните правой кнопкой мыши и из всплывшего меню выберите раздел **New Field**. Перед вами откроется окно создания нового поля, с которым вы уже знакомы по созданию вычисляемых полей (рис. 9.45). Введите имя (**Name**) создаваемого поля. Оно должно отличаться от имен других полей. Укажите тип (**Type**) создаваемого поля. Он, как правило, должен соответствовать типу того поля в таблице просмотра, которое вы хотите просматривать. В нашем примере вы хотите просматривать в таблице **Dep** поле **Proizv**, имеющее булев тип. Для строк и некоторых других типов полей надо еще указать размер поля (**Size**).

Рис. 9.45.
Ввод информации о поле просмотра



После того, как вы определили новое поле, нажмите в группе радиокнопок **Field type** кнопку **Lookup**. Выберите в выпадающем списке **Key Fields** ключевое поле (или поля) в таблице, в которой вы создаете новое поле (**Table2**). Это то поле или поля, по которым вы будете связываться с другой таблицей. В выпадающем списке **Dataset** выберите таблицу, которую вы хотите просматривать (**Table1**). Далее в выпадающем списке **Lookup Keys** выберите ключевое поле (или поля) просматриваемой таблицы (в нашем примере поле **Dep** из **Table1**). И, наконец, в выпадающем списке **Result Field** выберите просматриваемое поле (**Proizv**). Щелкните на **OK**. Вы вернетесь в Редактор Полей, в котором появится ваше новое поле **Pro**. Вам осталось задать для него в Инспекторе Объектов свойство **DisplayLabel** (заголовок), например, «Тип», и, поскольку это поле булево, задать свойство **DisplayValues** (отображаемые значения для **true** и **false**), например, «произв.;управл.».

Теперь в ваших компонентах визуализации данных, например, в **DBGrid**, появится соответствующее поле (см. рис. 9.46).

Рис. 9.46.
Приложение с введенными полями просмотра отдела и типа отдела (после столбца «Отдел»)

Фамилия	Имя	Отчество	Отдел	Тип	г.р.	Пол
Андреев	Андрей	Андреевич	Цех 2	произв.	1930	м
Антонсва	Антонина	Антоновна	Бухгалтерия	управл.	1965	ж
Борисов	Борис	Борисович	Цех 1	произв.	1937	м
Иванников	Иван	Иванович	Бухгалтерия	произв.	1975	м
Иванов	Иван	Иванович	Цех 1	управл.	1950	м
Иванова	Ирина	Ивановна	Цех 1	произв.	1971	ж

Обратите внимание, что для введенного вами поля **Pro** в Инспекторе Объектов определены свойства **FieldKind** (с установленным значением **fkLookup**), **KeyFields**, **LookupDataset**, **LookupKeyField** и **LookupResultField**. В принципе вы могли бы задать все эти свойства непосредственно в Инспекторе Объектов, но использование Редактора Полей более удобно.

В разделе 9.7 говорилось, что для полей просмотра в **DBGrid** можно в свойстве **Columns** задать свойство **ButtonStyle** поля равным **cbsAuto** и тогда при редактировании данных автоматически будет появляться выпадающий список, из которого пользователь может выбирать соответствующее значение. В нашем примере хорошо было бы это сделать для поля с названием отдела. Но ведь такое поле уже есть и оно не просматриваемое. Как тут быть? Эту сложность можно обойти следующим образом. Введите так, как было рассказано ранее, новое поле просмотра, назвав его, например, **DepLook** и задав для него в окне рис. 9.45 значение **Result Field** равным **Dep**. Для вашего нового поля в Редакторе Полей задайте те же свойства (прежде всего — **DisplayLabel**), какие задавали ранее для поля **Dep**. А затем в свойстве **Columns** компонента **DBGrid** удалите поле **Dep** и вместо него вставьте в таблицу поле **DepLook**. Ваша таблица будет выглядеть так же, как и ранее. Но при редактировании отдела в какой-нибудь записи в соответствующей ячейке таблицы будет появляться выпадающий список с названиями отделов. На рис. 9.46 изображен именно такой момент, когда сотрудника Борисова переводят из цеха 1 в цех 2.

Остановимся коротко еще на одном свойстве полей просмотра — **LookupCache**. Это свойство определяет, будут ли значения просматриваемого поля кэшироваться, или просмотр будет осуществляться при каждом изменении текущей записи. Если просматриваемая таблица (та, из которой берутся данные) не изменяется или изменяется редко, лучше задать **LookupCache = true**. Это существенно сократит затраты времени на просмотр, который будет осуществляться не в таблице, а в кэше, загружаемом один раз при открытии базы данных. Но если в процессе работы просматриваемая таблица изменяется, а **LookupCache = true**, то результат просмотра может быть неверным. В этих случаях надо вызывать метод **RefreshLookupList**, чтобы обновить хранящийся список.

Поля просмотра определяются до расчета вычисляемых полей в той же записи. Поэтому их значения можно использовать в вычисляемых полях, но не наоборот.

Поля просмотра имеют еще одну интересную особенность. Дело в том, что для них предусмотрены специальные компоненты: **DBLookupListBox** и **DBLookupComboBox** — список и выпадающий список. Если вы поместите на форму один из этих компонентов, установите у него свойство **DataSource**, соответствующее таблице, имеющей поля просмотра, а в свойстве **DataField** выберете одно из полей просмотра, то значения просматриваемого поля сразу отобразятся в списке. При этом не надо заботиться о загрузке в список значений, как это приходится делать со списками **DBListBox** и **DBComboBox**.

9.11 Программирование работы с базами данных

9.11.1 Состояние набора данных

В рассмотренных ранее простых приложениях практически не требовалось программирование. Все сводилось к размещению на форме компонентов и заданию их свойств. Но более изощренные приложения все-таки требуют программирования и написания различных обработчиков событий.

Начнем рассмотрение вопросов программирования работы с базами данных с основного свойства **State** компонента типа **TTable**, определяющего состояние набора данных. Это свойство доступно только во время выполнения и только для чтения. Набор данных может находиться в одном из следующих основных состояний:

dsInactive	Набор данных закрыт, данные недоступны
dsBrowse	Данные могут наблюдаться, но не могут изменяться. Это состояние по умолчанию после открытия набора данных
dsEdit	Текущая запись может редактироваться
dsInsert	Может вставляться новая запись
dsSetKey	Доступен режим поиска записи и операция задания диапазона изменений SetRange . Может наблюдаться только ограниченное множество данных и не может проводиться редактирование или вставка новой записи

Состояния могут устанавливаться в приложении во время выполнения, но не непосредственно, а с использованием различных методов.

Метод **Close** закрывает соединение с базой данных, устанавливая свойство **Active** набора данных в **false**. При этом **State** переводится в состояние **dsInactive**.

Метод **Open** открывает соединение с базой данных, устанавливая свойство **Active** набора данных в **true**. При этом **State** переводится в состояние **dsBrowse**.

Метод **Edit** переводит набор данных в состояние **dsEdit**.

Методы **Insert** и **InsertRecord** вставляют новую пустую запись в набор данных, выполняют еще ряд операций, о которых вы можете посмотреть в справке **C++Builder**, и переводят **State** в состояние **dsInsert**.

Методы **EditKey**, **SetRange**, **SetRangeStart**, **SetRangeEnd** и **ApplyRange**, связанные с поиском записи и с заданием допустимого диапазона изменения данных, переводят **State** в состояние **dsSetKey**. Эти методы будут рассмотрены позднее.

Многие другие методы также устанавливают автоматически различные состояния набора данных. При программировании работы с базой данных надо следить за тем, чтобы набор данных вовремя был переведен в соответствующее состояние. Например, если вы стали редактировать запись, не переведя предварительно набор данных в состояние **dsEdit** методом **Edit**, то будет сгенерировано исключение **EDatabaseError** с сообщением «Dataset not in edit or insert mode» — «Набор данных не в режиме Edit или Insert».

9.11.2 Пересылка записи в базу данных

Пока идет редактирование текущей записи, изменения осуществляются в буфере, а не в самой базе данных. Пересылка записи в базу данных производится только при выполнении метода **Post**. Метод может вызываться только тогда, когда набор данных находится в состоянии **dsEdit** или **dsInsert**. Этот метод можно вызывать явно, например, **Table1->Post()**. Но кроме того он вызывается неявно при любых перемещениях по набору данных, т.е. при перемещении курсора на новую текущую запись, если набор данных находится в состоянии **dsEdit** или **dsInsert**. Отменить изменения, внесенные в запись, можно методом **Cancel**. При выполнении этого метода, если предварительно не был вызван метод **Post**, запись возвращается к состоянию, предшествовавшему редактированию, и набор данных переводится в состояние **dsBrowse**.

В ряде случаев плохо то, что метод **Post** вызывается неявно при перемещениях по базе данных. Пользователь может по ошибке сделать неверные исправления в какой-то записи и, переместившись на другую запись, невольно занести их в таблицу без какой-то предварительной проверки. Надо принимать меры, чтобы без проверки данных и без подтверждения пользователем в базу данных ничего не заносилось. Это можно сделать, используя множество событий компонента **Table**.

Перед началом выполнения каждого из рассмотренных выше методов и после его выполнения генерируются соответствующие события набора данных **Table**. Например, перед выполнением метода **Post** возникает событие **BeforePost**, а после его окончания — событие **AfterPost**. Аналогичные события **BeforeInsert** и **AfterInsert** сопровождают выполнения метода **Insert** и т.п. Подобные события и надо использовать для проверки данных и получения подтверждения на их изменение.

Один из множества возможных вариантов заключается в использовании события **BeforePost**. Обработчик этого события может иметь вид:

```
void __fastcall TForm1::Table1BeforePost(TDataSet *DataSet)
{
    if (проверка введенных данных)
    {
        if (Application->MessageBox(
            "Хотите занести текущую запись в базу данных?",
            "Подтвердите занесение в базу данных",
            MB_YESNOCANCEL + MB_ICONQUESTION) != IDYES)
        {
            DataSet->Cancel();
            Abort();
        }
    }
    else
    {
        Application->MessageBox("Ошибочные данные", "Ошибка",
            MB_ICONSTOP);
        Abort();
    }
}
```

К этому обработчику будет происходить обращение перед выполнением метода **Post**, как бы он не был вызван: явно или вследствие перемещения по базе данных, если текущая запись была изменена. В обработчике сначала производится проверка данных в записи. Если она дает неудовлетворительный результат, то пользователю дается сообщение об ошибочности данных и выполняется функция **Abort**, прерывающая выполнение **Post**. Текущая запись остается в состоянии **dsEdit**, но ошибочные данные в ней не сбрасываются, что позволяет пользователю исправить их.

Если проверка данных в записи показала их правильность, то у пользователя запрашивается подтверждение изменений в базе данных. Если он ответит отрицательно, то для набора данных выполняется метод **Cancel**, а затем выполняется функция **Abort**. **Cancel** возвращает данные в текущей записи к состоянию, которое было до их редактирования, т.е. уничтожает результаты редактирования. В операторе **Cancel** использована ссылка **DataSet**. Это параметр, передаваемый в обработчик события **BeforePost** и соответствующий набору данных, к которому применяется **Post**. Использование параметра **DataSet** позволяет написать обработчик в более общем виде и к такому обработчику можно обращаться при событиях в разных наборах данных. Для конкретизации вместо **DataSet** можно было бы использовать имя конкретного набора данных, например, **Table1**.

Таким образом, приведенный выше обработчик события **BeforePost** позволяет предотвратить произвольное или ошибочное изменение данных.

Часто безопасность данных обеспечивается несколько иначе: при переходе в режим редактирования данные из текущей записи переносятся в какие-то обычные окна редактирования, не связанные с данными, там проводится их изменение и после того, как проведены все проверки и получено подтверждение пользователя на их сохранение, эти данные переносятся в таблицу. Но для выполнения подобных операций необходимо иметь доступ к отдельным полям записи. Позднее мы посмотрим, как это можно делать.

9.11.3 Кэширование изменений

По умолчанию все изменения в наборе данных, завершаемые методами **Post**, **Insert**, **Delete** и т.п., заносятся в базу данных. Однако, возможен и другой режим работы, при котором все изменения, вносимые пользователем в записи, кэшируются — т.е. сохраняются в памяти локально. В этом случае пользователь работает не с реальными данными, а с их копиями. И только по специальной команде все внесенные пользователем изменения заносятся в базу данных.

Режим кэширования определяется свойством **CachedUpdates** компонента **Table**. По умолчанию это свойство равно **false** и кэширование не производится. Если установить его в **true**, то все изменения набора данных будут кэшироваться. Они передаются в базу данных только после выполнения метода **ApplyUpdates**. Если же после множества изменений выполнить метод **CancelUpdates**, то все изменения, произведенные после последнего выполнения **ApplyUpdates**, отменяются.

Метод **ApplyUpdates** только передает изменения в базу данных на сохранение, но еще не фиксирует изменения данных. Метод **CommitUpdates**, который должен выполняться после **ApplyUpdates**, очищает буфер кэша, после чего он готов для приема новой порции информации, а измененные данные фиксируются в базе данных. Если при фиксации данных произошла ошибка, можно применить метод **Rollback**, который аннулирует все изменения.

Проверьте все это, построив простое приложение, вид которого показан на рис. 9.47. Индикатор Кэширование переключает режим кэширования. Кнопка Фиксация фиксирует все сделанные изменения в базе данных. Кнопка Отмена отменяет сделанные изменения. Когда приложение закрывается, надо проверить, не работало ли оно в режиме кэширования и не было ли сделано изменений, которые не зафиксированы в базе данных. Если были, то следует спросить пользователя о необходимости их сохранения и при положительном ответе зафиксировать изменения.

Коды этого приложения очень простые. Для таблицы **Table1** свойство **CachedUpdates** первоначально надо задать равным **true**. В приложение можно ввести переменную **modif**, которая будет фиксировать наличие несохраненных изменений:

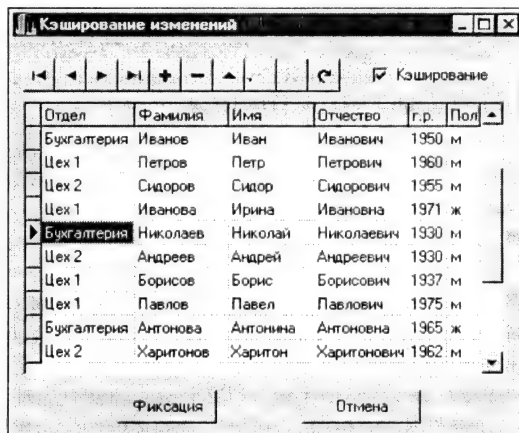
```
bool modif = false;
```

Для события **OnClick** индикатора Кэширование надо предусмотреть обработчик:

```
Table1->CachedUpdates = ! Table1->CachedUpdates;  
BApplyUpdates->Enabled = Table1->CachedUpdates;  
BCancelUpdates->Enabled = Table1->CachedUpdates;  
if (Table1->CachedUpdates) modif = false;
```

Рис. 9.47.

Приложение, демонстрирующее кэширование данных



Он изменяет режим кэширования таблицы, делает доступными или недоступными в зависимости от режима кнопки Фиксация (ее имя — **BApplyUpdates**) и Отмена (ее имя — **BCancelUpdates**) и, если включается режим кэширования, то сбрасывается в **false** значение **modif**.

Для событий **AfterEdit**, **AfterDelete** и **AfterInsert** компонента **Table1** можно предусмотреть обработчик:

```
if (Table1->CachedUpdates) modif = true;
```

который фиксирует в переменной **modif** факт редактирования, удаления или вставки записи.

Обработчик события **OnClick** кнопки Фиксация имеет вид

```
Table1->ApplyUpdates();
Table1->CommitUpdates();
modif = false;
```

Обработчик события **OnClick** кнопки Отмена имеет вид

```
Table1->CancelUpdates();
modif = false;
```

В обоих обработчиках переменная **modif** сбрасывает в **false**, фиксируя отсутствие несохраненных изменений.

Обработчик события **OnCloseQuery** формы имеет вид:

```
if (Table1->CachedUpdates && modif)
    switch (Application->MessageBox(
        "Сохранить изменения в базе данных?",
        "Подтвердите изменения",
        MB_YESNOCANCEL + MB_ICONQUESTION))
    {
        case IDYES:    Table1->ApplyUpdates();
                       break;
        case IDCANCEL: CanClose = false;
                       break;
        case IDNO:     Table1->CancelUpdates();
    }
```

При наличии не сохраненных изменений пользователю задается вопрос «Сохранить изменения в базе данных?». При положительном ответе изменения сохраняются методом **ApplyUpdates**. При отрицательном — изменения отменяются методом **CancelUpdates**. Если при получении запроса пользователь нажал кнопку Отмена, то приложение не закрывается (**CanClose** задается равным **false**).

Сделайте такое приложение и поработайте с ним, чтобы почувствовать различие в работе при наличии и отсутствии кэширования.

9.11.4 Доступ к полям

Поля отображаются объектами класса **TField** и производных от него классов **TStringField**, **TSmallintField**, **TBooleanField** и т.п. Эти объекты могут создаваться тремя способами:

- Автоматически генерироваться для каждого компонента набора данных (**Table** и др.)
- Создаваться в процессе проектирования с помощью Редактора Полей
- Создаваться программно в процессе выполнения приложения

Автоматическая генерация объектов класса **TField** происходит в момент открытия базы данных, если эти объекты не создаются другими способами: в процессе проектирования с помощью Редактора Полей или программно во время выполнения. Объекты генерируются для всех полей таблицы (это может быть избыточно для конкретного приложения).

Создание объектов полей в процессе проектирования с помощью Редактора Полей было рассмотрено ранее. Этот вариант исключает автоматическое создание объектов. Таким образом, все поля, объекты которых не созданы с помощью Редактора Полей, недоступны для приложения. Использование Редактора Полей, как мы видели, очень удобно, так как позволяет задать в процессе проектирования множество полезных свойств для каждого поля.

Программное создание объектов класса **TField** используется сравнительно редко и мы его рассматривать не будем.

Доступ к объектам полей возможен тремя способами:

- По порядковому индексу объекта
- По имени поля
- По имени объекта

Доступ по порядковому индексу осуществляется через свойство **TField* Fields[int i]**, где **i** — индекс объекта. Индексация, как всегда в C++Builder, начинается с 0. Например, **Table1->Fields[0]** — это первый объект поля таблицы **Table1**.

Доступ по имени поля осуществляется с помощью метода **FieldByName(«имя»)**. Например, **Table1->FieldByName(«Fam»)** — это объект, связанный с полем **Fam**.

Доступ по имени объекта возможен только к объектам, созданным с помощью Редактора Полей. По умолчанию C++Builder формирует имена объектов полей (**Name**) из имени таблицы и имени поля. Например, **Table1Dep**. Вы можете видеть эти имена, работая с Редактором Полей. Конечно, вы можете изменить это имя на любое другое. Обращение к объекту по имени не требует, в отличие от предыдущих вариантов, ссылки на таблицу. Вы можете просто написать **Table1Dep** и это будет необходимый вам объект.

Автоматически создаваемые объекты имени не имеют — их свойство **Name** пусто. Поэтому для них обращение по имени невозможно.

Среди рассмотренных способов доступа к полям наиболее быстрым, конечно, является доступ по имени объекта. Его недостатком является жесткая кодировка поля, к которому производится обращение. Если надо, чтобы строка кода в разных ситуациях обращалась к разным полям, то надо использовать или доступ по индексу **Fields[i]** (тогда индекс **i** можно сделать переменным), или по имени поля методом **FieldByName(s)** (строку **s** можно сделать переменной).

Вы уже видели множество свойств объектов класса **TField** и производных от них классов. Это свойства **ReadOnly**, **DisplayLabel**, **CustomConstraint** и многие другие. Сейчас рассмотрим, как добраться до главного свойства объекта — хранящегося в нем значения поля текущей записи.

Значение поля хранится в свойстве **Value**. Тип этого свойства — **Variant**, т.е. тип определяется типом поля. Например, **Table1->FieldByName(«Fam»)->Value** — это строка, а **Table1->FieldByName(«Year_b»)->Value** — это целое число.

Имеется также ряд свойств, переводящих один тип значений в другой. Например, свойство **AsString** переводит тип любого поля в строку при чтении значения поля, и осуществляет обратный перевод строки в тип поля при записи значения поля. Вы можете написать

```
EDep->Text = Table1->FieldByName("Dep")->AsString;  
EYear->Text = Table1->FieldByName("Year_b")->AsString;  
ESex->Text = Table1->FieldByName("Sex")->AsString;
```

и в окна редактирования **EDep**, **EYear** и **ESex** будут занесены в текстовом виде значения в текущей записи полей **Dep**, **Year_b** и **Sex**, хотя поле **Dep** имеет тип строки, поле **Year_b** — целое значение, а поле **Sex** — булево. Если для поля **Sex** вы не задавали значений **DisplayValues**, то в окне редактирования **ESex** будут отображены значения «true» или «false». Если же вы задали значения свойства **DisplayValues**, например «м;ж» или «мужск;женск», то отобразятся именно эти заданные значения.

То же свойство **AsString** работает и как обратное преобразование типов. Продолжая предыдущий пример вы можете после того, как пользователь отредактировал значения в полях редактирования **EDep**, **EYear** и **ESex**, внести эти значения в текущую запись, например, следующим кодом:

```
Table1->Edit();
Table1->FieldByName("Dep")->AsString = EDep->Text;
Table1->FieldByName("Year_b")->AsString = EYear->Text;
Table1->FieldByName("Sex")->AsString = ESex->Text;
Table1->Post();
```

Для полей **Year_b** и **Sex** текст будет преобразован соответственно в целое и булево значение. При этом не обязательно, чтобы пользователь в окне **ESex** писал полностью обозначение пола сотрудника. Ему достаточно написать только первую букву: «t» или «f», если отображаемые значения **true** и **false**, и «м» или «ж», если отображаемые значения «мужск» и «женск».

Описанный выше способ, заключающийся в использовании для ввода и отображения данных компонентов (в этом примере — **Edit**), не связанных непосредственно с данными, — один из наиболее надежных для редактирования записи в базе данных. Прежде, чем вносить отредактированные данные в таблицу, вы можете их всесторонне проверить, запросить у пользователя подтверждение записи и только после этого заносить изменения в базу данных. При этом вы в процессе редактирования и проверки не связаны ни с какими событиями и состояниями базы данных.

Помимо **AsString** имеются еще аналогичные свойства **AsInteger**, **AsFloat**, **AsBoolean**, **AsDateTime**. Свойство **AsInteger** осуществляет преобразования между типом данного поля и целым 32-битным числом, свойство **AsFloat** делает то же самое для действительных чисел с плавающей запятой, свойство **AsBoolean** — для булевых значений, свойство **AsDateTime** — для значений дат и времени в принятом в C++Builder формате **TDateTime**.

В некоторых случаях вам может потребоваться узнать тип данного поля. Вы можете это узнать из свойства объекта поля **DataType**, которое может принимать значения: **ftUnknown** (неизвестное), **ftAutoInc** (автоматически нарастающее), **ftString** (строка) и т.д.

9.11.5 Методы навигации

Наборы данных имеют ряд методов, позволяющих осуществлять навигацию — перемещение по таблице:

First	Перемещение к первой записи
Last	Перемещение к последней записи
Next	Перемещение к следующей записи
Prior	Перемещение к предыдущей записи
MoveTo(int i)	Перемещение к концу (при $i > 0$) или к началу ($VxP\ i < 0$) на i записей

Любое перемещение по набору, находящемуся в состоянии **dsEdit**, автоматически вызывает метод **Post**, который пересылает текущую запись, если она была изменена, в базу данных. И после этого уже невозможно применить метод **Cancel** для уничтожения изменений. Поэтому надо принимать меры, чтобы в наборе данных, находящемся в состоянии **dsEdit**, перед любым перемещением происходила проверка правильности данных (в разделе 9.11.2 было показано, как это можно сделать), или перемещаться по набору в каком-то другом состоянии (например, в **dsBrowse**).

При перемещениях можно совершить ошибку, выйдя за пределы имеющихся записей. Например, если вы находитесь на первой записи и выполняете метод **Prior**, то вы выйдете за начало таблицы, а если вы находитесь на последней записи и выполняете метод **Next**, то вы окажетесь после последней записи. Чтобы контролировать начало и конец таблицы, существуют два свойства: **Eof** (end-of-file) — конец данных, и **Bof** (beginning of file) — начало данных. Эти свойства становятся равными **true**, если делается попытка переместить курсор соответственно за пределы последней или первой записи, а также после выполнения методов соответственно **Last** и **First**.

Приведем пример. Пусть в вашем приложении имеется выпадающий список с именем **CBdep**, который вы хотите заполнить данными, содержащимися в полях **Dep** всех записей таблицы, соединенной с компонентом **Table1**. Это можно сделать следующим кодом:

```
CBdep->Clear();
Table1->First();
while (!Table1->Eof)
{
    CBdep->Items->Add(Table1->AsString);
    Table1->Next();
}
CBdep->ItemIndex = 0;
Table1->First();
```

Первый оператор кода очищает список **CBdep**. Второй — устанавливает курсор таблицы на первую запись. Далее следует цикл по всем записям, пока не достигнута последняя, что проверяется выражением **Table1->Eof**. Для каждой записи в список заносится значение поля **Dep**, после чего методом **Next** курсор перемещается к следующей записи. После окончания цикла индекс списка и курсор таблицы переводятся соответственно на первую строку и запись.

9.11.6 Поиск записей

Одна из важнейших для пользователя операций с базами данных — поиск записей по некоторому ключу. Существует несколько методик поиска записей, которые можно назвать **SetKey**, **FindKey**, **Lookup** и **Locate**.

Начнем с методики **SetKey**. Для ее применения таблица предварительно должна быть индексирована по тому полю, по которому должен будет проводиться поиск. Затем таблица устанавливается в состояние поиска **dsSetKey**. Для этого используется метод **SetKey**. В состоянии **dsSetKey** набор данных воспринимает последующий оператор присваивания значения полю не как присваивание, а как задание ключа поиска. Поэтому после установки состояния **dsSetKey** оператором присваивания устанавливается требуемое значение ключа поиска по интересующему полю. В заключение методом **GotoKey** курсор переводится на запись, в которой значение указанного поля равно ключу. Если таких записей несколько, то курсор переводится на первую из них. Если соответствующая запись не находится, то метод **GotoKey** возвращает **false**. Для полей типа строк лучше использовать не метод **GotoKey**, а метод **GotoNearest**. Этот метод перемещает курсор на первую запись, значение поля в которой максимально близко к ключу. Т.е. он сработает и тогда, когда совпадение не полное. Метод **GotoNearest** можно применять и к цифровым полям. В этом случае он переместит курсор на первую запись, значение поля в которой больше или равно заданному значению ключа.

Например, если вы хотите найти первую запись, в которой год рождения (поле **Year_b**) равен заданному пользователем в окне редактирования **EYear**, вы можете написать операторы:

```
Table1->IndexFieldNames = "Year_b";
Table1->SetKey();
```

```
Table1->FieldByName("Year_b")->AsString = EYear->Text;
if (! Table1->GotoKey())
    ShowMessage("Запись не найдена");
```

Первый оператор индексирует набор данных по полю `Year_b`, второй переводит набор данных в состояние `dsSetKey`, третий задает ключ поиска, а четвертый осуществляет переход к соответствующей записи или сообщает об отсутствии такой записи.

Если вы хотите найти в таблице сотрудника по его фамилии, заданной пользователем в окне редактирование `EFam`, вы можете выполнить код:

```
Table1->IndexFieldNames = "Fam";
Table1->SetKey();
Table1->FieldByName("Fam")->AsString = EFam->Text;
Table1->GotoNearest();
```

Даже если точно такой фамилии не найдется, курсор перейдет на наиболее похожую (совпадающую по первым символам).

Методика поиска `FindKey` еще богаче по своим возможностям. В этой методике таблица также должна быть проиндексирована по тем ключевым полям, по которым осуществляется поиск. Функция `FindKey` определена следующим образом:

```
bool __fastcall FindKey(const System::TVarRec * KeyValues,
    const int KeyValues_Size);
```

Параметр `KeyValues` представляет собой открытый массив: разделяемый запятыми список значений полей, по которым индексирован набор данных, в той последовательности, в которой они входят в индекс. При этом не обязательно перечислять все поля — достаточно перечислить первое или несколько первых. Параметр `KeyValues_Size` определяет индекс последнего поля в массиве, участвующего в поиске. Поскольку индексы начинаются с 0, то `KeyValues_Size` на единицу меньше количества полей, участвующих в поиске.

Вместо `FindKey` для полей строкового типа можно использовать аналогичный метод `FindNearest`, обеспечивающий переход к наиболее совпадающей строке, если полного совпадения не получено. Объявление и параметры этого метода те же, что и в методе `FindKey`. Метод `FindNearest` можно применять и к цифровым полям. В этом случае он переместит курсор на первую запись, значение полей в которой больше или равны заданных значений ключей.

Для методов `FindKey` и `FindNearest` удобно применять макрос `OPENARRAY`, создающий временный открытый массив и определяющий его размер:

```
OPENARRAY(TVarRec, (список ключей))
```

Макрос `OPENARRAY` может воспринимать список, включающий до 19 элементов, разделенных запятыми.

Рассмотрим примеры. Пусть мы хотим выполнить тот же поиск по фамилии, что и приведенный выше. В данном случае соответствующий код может иметь вид:

```
Table1->IndexFieldNames = "Fam";
Table1->FindNearest(&TVarRec(EFam->Text), 0);
```

Если мы хотим выполнить аналогичный поиск, но нас интересует не просто один из сотрудников с заданной фамилией, а работающий в конкретном подразделении, заданном пользователем в окне редактирования `EDep`, то поиск можно осуществить с помощью макроса `OPENARRAY` следующим образом:

```
Table1->IndexFieldNames = "Dep;Fam";
Table1->FindNearest(OPENARRAY(TVarRec, (EDep->Text, EFam->Text)));
```

Первый оператор индексирует таблицу по полям `Dep` и `Fam`, а второй задает ключи для этих полей. Конечно, не надо индексировать таблицу каждый раз перед осуществлением поиска. Достаточно выполнить это один раз.

Приведем еще один пример. Пусть мы хотим предоставить пользователю ускоренный поиск фамилии. Пользователь будет набирать фамилию в окне **EFam** и при вводе каждого нового символа курсор должен перемещаться на запись, наиболее близко совпадающую с уже введенными символами. Для этого сначала надо индексировать таблицу по полю **Fam**. А затем достаточно в событие **OnChange** окна редактирования **EFam** вставить обработчик

```
Table1->FindNearest(&TVarRec(EFam->Text),0);
```

Теперь остановимся на методе **Locate**. Этот метод объявлен следующим образом:

```
bool __fastcall Locate(const System::AnsiString KeyFields,
                      const System::Variant &KeyValues,
                      TLocateOptions Options);
```

В качестве первого параметра **KeyFields** передается строка, содержащая список ключевых полей. В качестве второго параметра передается **KeyValues** — массив ключевых значений. А третий параметр **Options** является множеством опций, элементами которого могут быть **loCaseInsensitive** — нечувствительность поиска к регистру, в котором введены символы, и **loPartialKey** — допустимость частичного совпадения. Метод возвращает **false**, если искомая запись не найдена.

В простейшем случае применение **Locate** отличается от рассмотренных ранее методов только отсутствием необходимости индексировать набор данных определенным образом. Например, поиск (в том числе и рассмотренный выше ускоренный поиск) по фамилии может осуществляться операторами

```
TLocateOptions SearchOptions;
SearchOptions << loPartialKey << loCaseInsensitive;
Table1->Locate("Fam",EFam->Text, SearchOptions);
```

причем он будет работать независимо от того, как индексирована база данных. Впрочем, если набор данных соответствующим образом индексирован, то поиск производится быстрее.

Приведенный код можно сократить до двух операторов:

```
TLocateOptions SearchOptions;
Table1->Locate("Fam", EFam->Text,
              SearchOptions<<loPartialKey<<loCaseInsensitive);
```

При поиске по нескольким полям можно воспользоваться функцией **VarArrayOf**, которая формирует тип **Variant** из задаваемого ей массива параметров любого типа. Например, рассмотренный ранее поиск по отделу и фамилии может быть осуществлен операторами

```
TLocateOptions SearchOptions;
Variant locvalues[] = {EDep->Text, EFam->Text};
Table1->Locate("Dep:Fam", VarArrayOf(locvalues,1),
              SearchOptions<<loPartialKey<<loCaseInsensitive);
```

И в заключение о последнем методе поиска — **Lookup**. Этот метод определен следующим образом:

```
System::Variant __fastcall Lookup(
    const System::AnsiString KeyFields,
    const System::Variant &KeyValues,
    const System::AnsiString ResultFields);
```

Первые два параметра аналогичны методу **Locate**. Третий параметр — строка, перечисляющая поля, значения которых возвращаются в виде массива **Variant**. Если не найдено соответствующей записи, функция возвращает **false**.

Например, если вы хотите найти запись, относящуюся к сотруднику, фамилия которого указана в окне **EFam**, и вывести в окно **EDep** название отдела, в котором он работает, то эти операции можно осуществить следующим образом:

```
EDep->Text = Table1->Lookup("Fam", EFam->Text, "Dep");
```

Метод **Lookup** не изменяет положения курсора. Он только возвращает значения указанных полей. Они могут использоваться любым способом. В частности, они могут использоваться вместо параметра **KeyValues** в другом операторе **Lookup** или **Locate**. Это открывает широкие возможности формирования сложных запросов по нескольким таблицам.

9.11.7 Методы установки диапазона допустимых значений

Ранее рассматривались различные способы задания критериев отбора установкой соответствующих значений свойств полей и наборов данных. Теперь рассмотрим коротко несколько методов, позволяющих задавать диапазон допустимых значений поля во время выполнения приложения. Метод **SetRangeStart** переводит набор данных в состояние **dsSetKey** и следующий оператор присваивания значения полю воспринимается как задание для поля нижней границы диапазона возможных значений. Метод **SetRangeEnd** действует аналогично, но последующий оператор присваивания воспринимается как задание верхней границы диапазона. После того, как пределы диапазона установлены, можно выполнить метод **ApplyRange**. При этом начнут использоваться установленные границы диапазона и доступными для просмотра и редактирования будут только те записи, в которых значения указанного поля находятся внутри диапазона.

Например, операторы

```
Table1->IndexFieldNames = "Fam";
Table1->SetRangeStart();
Table1->FieldByName("Fam")->AsString = "A";
Table1->SetRangeEnd();
Table1->FieldByName("Fam")->AsString = "I";
Table1->ApplyRange();
```

приведут к тому, что доступными будут только записи сотрудников, фамилии которых начинаются с букв «А», «Б», «В».

На результаты работы методов **SetRangeStart** и **SetRangeEnd** для числовых полей влияет свойство набора данных **KeyExclusive**. Оно определяет, будут ли считаться сами заданные границы входящими в диапазон (при **KeyExclusive = false**, это значение принято по умолчанию), или не входящими (при **KeyExclusive = true**). Иначе говоря, при **KeyExclusive = false** используются операции отношения **i, j**, а при **KeyExclusive = true** — **>, <**. Например, если вы хотите отобразить записи, в которых год рождения сотрудников лежит в определенных пределах, то при коде

```
Table1->IndexFieldNames = "Year_b";
Table1->SetRangeStart();
Table1->FieldByName("Year_b")->AsInteger = 1950;
Table1->SetRangeEnd();
Table1->FieldByName("Year_b")->AsInteger = 1960;
Table1->ApplyRange();
```

будут отобраны записи, в которых год рождения лежит в пределах 1950 — 1960, включая годы 1950 и 1960. А при коде

```
Table1->IndexFieldNames = "Year_b";
Table1->SetRangeStart();
Table1->FieldByName("Year_b")->AsInteger = 1950;
Table1->KeyExclusive = true;
Table1->SetRangeEnd();
Table1->FieldByName("Year_b")->AsInteger = 1960;
Table1->ApplyRange();
```

записи, соответствующие 1950 году, не войдут в число отобранных.

Имеется и более простой способ установки диапазона — метод **SetRange**. Он определен следующим образом:

```
void __fastcall SetRange(const System::TVarRec * StartValues,
                        const int StartValues_Size,
                        const System::TVarRec * EndValues,
                        const int EndValues_Size);
```

Открытые массивы **StartValues** и **EndValues** должны содержать соответственно нижние и верхние значения диапазонов полей, являющихся ключевыми. Таким образом, метод **SetRange** заменяет последовательное обращение к методам **SetRangeStart**, **SetRangeEnd** и **ApplyRange**. Например, приведенные ранее операторы, задающие диапазон фамилий в отобранных записях, могут быть заменены следующими:

```
Table1->IndexFieldNames = "Fam";
Table1->SetRange(&TVarRec("A"), 0, &TVarRec("Г"), 0);
```

Можно, конечно, использовать и другие способы работы с открытыми массивами типа **TVarRec** (в частности, макрос **OPENARRAY**), рассмотренные в разделе 9.11.6.

Если вы хотите работать не просто со всеми сотрудниками, фамилии которых начинаются на «А» — «В», а с теми из них, которые работают в цехе 1, то приведенные выше операторы можно изменить следующим образом:

```
Table1->IndexFieldNames = "Dep;Fam";
Table1->SetRange(OPENARRAY(TVarRec, ("Цех 1", "A")),
                OPENARRAY(TVarRec, ("Цех 1", "Г")));
```

9.11.8 Методы модификации таблиц

Помимо рассмотренных ранее методов модификации записей **Table** имеет также ряд методов, позволяющих модифицировать таблицы и индексы. Коротко перечислим некоторые из них.

CreateTable	Метод создает новую таблицу, исходя из установок компонента Table , содержащихся в свойствах Fields или FieldDefs . Если таблица с именем, указанным в свойстве TableName уже имеется, она будет переписана. Применение этого метода позволяет, например, взять структуру существующей таблицы, как-то изменить ее, затем изменить свойство TableName на имя новой таблицы и создать эту таблицу
DeleteTable	Метод уничтожает существующую таблицу, которая задана свойствами DatabaseName и TableName . Таблицу надо предварительно закрыть
RenameTable(s)	Метод переименовывает существующую таблицу, присваивая ей новое имя, содержащееся в s . Одновременно переименоваются все сопутствующие таблице файлы
DeleteIndex(s)	Удаляет вторичный индекс с именем s из таблицы

Ниже в качестве примера приведен код, создающий используемую в данной книге таблицу **Dep**. Предполагается, что в приложении имеется компонент **Table1**, связанный с базой данных, в которой создается таблица.

```
// Компонент Table1 делается неактивным
Table1->Active = False;
// Указывается имя таблицы
```

```

Table1->TableName = "Dep.db";
// Таблица создается, если в базе данных нет такой таблицы
if(! Table1->Exists)
{
    // Указывается тип таблицы
    Table1->TableType = ttParadox;
    // Начало описания полей таблицы
    Table1->FieldDefs->Clear();
    // Создается указатель на объект описания поля
    TFieldDef *pNewDef = Table1->FieldDefs->AddFieldDef();
    // Описание первого поля
    pNewDef->Name = "Dep";
    pNewDef->DataType = ftString;
    pNewDef->Size = 20;
    pNewDef->Required = True;
    // Описание второго поля
    pNewDef = Table1->FieldDefs->AddFieldDef();
    pNewDef->Name = "Proisv";
    pNewDef->DataType = ftBoolean;

    // Описание индекса
    Table1->IndexDefs->Clear();
    // Индекс без имени - первичный ключ таблицы
    Table1->IndexDefs->Add("", "Dep",
        TIndexOptions() <<ixPrimary << ixUnique);

    // Создание таблицы методом CreateTable
    Table1->CreateTable();
    Table1->Open();
    // Вставка первой записи
    Table1->Insert();
    Table1->FieldByName("Dep")->AsString = "Бухгалтерия";
    Table1->FieldByName("Proisv")->AsBoolean = false;
    Table1->Post();
    ...

```

9.11.9 Модули данных

При разработке сложных приложений, работающих с базами данных, принято разделять логику работы и пользовательский интерфейс. В C++Builder это помогают сделать модули данных — компоненты контейнеры типа **TDataModule**.

В C++Builder 5 для проектирования модулей данных появился очень удобный инструмент — Проектировщик Модулей Данных (Data Module Designer). Он вызывается командой File | New и выбором в окне Депозитария на странице New пиктограммы Data Module. В результате открывается диалоговое окно, представленное на рис. 9.48 и 9.49. В левой панели по мере проектирования модуля данных вы можете видеть дерево помещенных вами в модуль компонентов наборов и источников данных, их полей, ограничений и т.п. Правая панель содержит две страницы: Components — компоненты (рис. 9.48), и Data Diagram — диаграмма данных (рис. 9.49).

Перенос в модуль новых компонентов осуществляется щелчком на соответствующей пиктограмме палитры библиотеки компонентов и последующим щелчком в левой панели Проектировщика Модулей Данных или на его странице Components. В результате в дереве и на странице Components появляются соответствующая вершина дерева и пиктограмма компонента.

В дереве или на странице Components вы можете получить доступ к любому компоненту или его полю. Выделив в дереве вершину какого-то компонента, например, таблицы **Table** (на рис. 9.48 и 9.49 компоненты **Table** названы **TDep** и **TPers**), вы увидите в Инспекторе объектов свойства и события этого компонента. Щелкнув на вершине компонента **Table** правой кнопкой мыши, вы можете вы-

Рис. 9.48.

Окно Проектировщика Модулей
Данных в С++Builder 5 с раскрытой
страницей компонентов

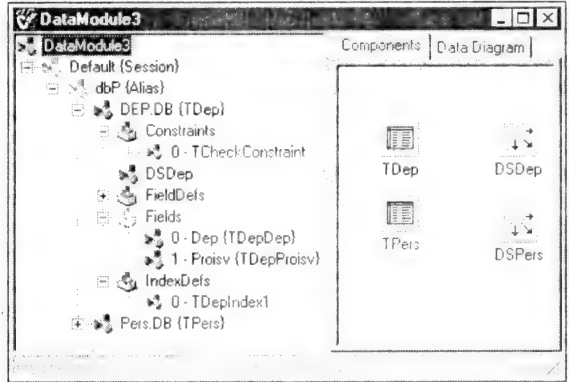
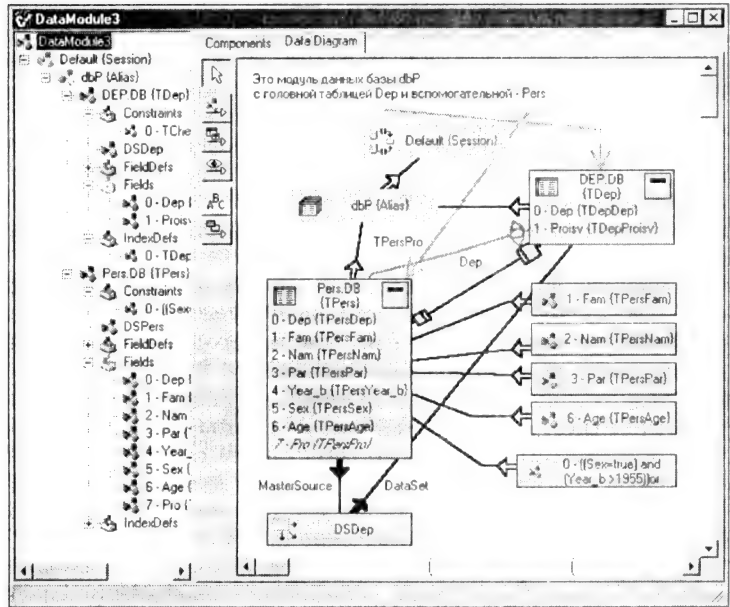


Рис. 9.49.

Окно Проектировщика
Модулей Данных в
С++Builder 5 с раскрытой
страницей диаграммы
данных



брать во всплывшем меню раздел Fields Editor — вызов Редактора Полей, и Explore — вызов SQL Explorer. Таким образом, вы можете выполнять манипуляции с полями, не покидая Проектировщик Модулей Данных.

Если вы щелкнете правой кнопкой мыши на вершине полей Fields, вы увидите другое контекстное меню, в котором можете выбрать разделы Add fields — добавить поля, Add all fields — добавить все поля, New field — создание нового поля, например, вычисляемого. Так что вы можете непосредственно из окна Проектировщика Модулей Данных выполнять основные команды Редактора Полей. Впрочем, и из этого контекстного меню вы можете вызвать непосредственно Редактор Полей командой Fields Editor. Выделив в дереве вершину какого-то поля, вы увидите в Инспекторе Объектов его свойства и события. Таким образом, все проектирование модуля данных и его компонентов осуществляется очень удобно и наглядно.

Вы можете перетаскивать в дереве вершины, изменяя таким образом информационные связи между компонентами. Например, вы можете перетаскать вершины компонентов **DataSource** (на рис. 9.48 и 9.40 они названы **DSDep** и **DSPers**) в ту или иную вершину набора данных, и они автоматически свяжутся с соответствующими компонентами, например, с **Table**. В их свойства **DataSet** при этом зане-

сутся соответствующие значения. Если же вы перетащите вершину источника данных в корневую вершину дерева, то источник данных не будет связан ни с каким набором данных.

Теперь рассмотрим страницу Data Diagram — диаграмма данных (рис. 9.49) правой панели Проектировщика Модулей Данных. На этой странице вы можете наглядно изобразить или спроектировать взаимоотношения между наборами данных, полями, ограничениями и т.п. Сначала рассмотрим возможности изображения связей, уже установленных между компонентами с помощью их свойств. Для этого достаточно перетащить мышью соответствующие вершины из панели дерева в правую панель. Связи между появившимися на панели блоками указываются автоматически. С помощью мыши вы можете разместить блоки должным образом на площади диаграммы и установить их размеры. Щелкнув на блоке правой кнопкой мыши, вы можете выбрать из контекстного меню раздел Color, позволяющий менять цвет фона компонента.

Если вы перетаскиваете из дерева вершину, имеющую дочерние вершины, и держите при этом нажатой клавишу Ctrl, то перетащатся и дочерние вершины, причем в диаграмме они расположатся по горизонтали. Если же при перетаскивании такой вершины вы держите нажатыми клавиши Ctrl и Shift, то вершины расположатся друг под другом.

В блоках, соответствующих наборам данных, вы можете видеть в правом верхнем углу пиктограммы с символом «-». Нажимая их вы можете сворачивать блоки, оставляя в них только заголовки, и разворачивать, в результате чего будут видны списки полей, введенных вами в Редакторе Полей (на рис. 9.49 блоки развернуты).

Изображения связей, возникающих на диаграмме, различаются в зависимости от типа связи. Первый тип связей указывает на соотношения между дочерними и родительскими блоками. Они изображаются контурными стрелками, идущими от дочернего блока к родительскому. На рис. 9.49 вы можете видеть подобные стрелки от полей к набору данных, от наборов данных к блоку базы данных и от базы данных к вершине Session. Эти типы связей отображают соотношения между родительскими и дочерними вершинами дерева и отображаются на диаграмме автоматически. Удалить их с диаграммы невозможно.

Второй тип связей обусловлен заданием тех или иных свойств компонентов. Такая связь отображается закрашенной стрелкой, около которой появляется надпись с именем свойства, посредством которого установлена связь. На рис. 9.49 вы можете видеть такие связи, входящие и выходящие из блока DSDep. Связь, около которой написано «DataSet», отображает связь между источником данных и набором данных, осуществленную свойством DataSet. А связь, около которой написано «MasterSource», отображает связь через свойство MasterSource вспомогательной таблицы TPers с главной таблицей TDep.

Если вы щелкнете на связи, основанной на значении свойства, правой кнопкой мыши, то можете выбрать из всплывшего меню раздел Remove Relationship — удалить связь. В результате связь разорвется и значение соответствующего свойства компонента очистится. Далее вы можете прямо на диаграмме построить вместо уничтоженной связи другую. Но об этом подробнее будет сказано ниже при рассмотрении способов проектирования модуля данных с помощью диаграммы.

Третий вид связей отображает связь главной таблицы со вспомогательной. На рис. 9.49 вы можете видеть такую связь, идущую из блока вспомогательной таблицы TPers в блок главной таблицы TDep. Около связи автоматически пишется имя полей, по которым осуществляется связь (на рис. 9.49 это поле Dep). Связь главной и вспомогательной таблицы можно разорвать приемом, аналогичным описанному для связей через свойства.

Четвертый вид связей отображает связь наборов данных через поля просмотра (см. раздел 9.10.2). На рис. 9.49 это связь между **TPers** и **TDep**, на которой схематически изображен глаз и около которой написано имя результирующего поля просмотра — **TPersPro**. Подобные связи можно разрывать так же, как описанные выше.

Местоположение и форму любой линии связи можно изменять. Если вы потянете курсором за один из концов связи, то можете переместить его, правда, в большинстве случаев не непрерывно, а дискретно. А если вы потянете за какую-то из средних точек линии, то линия получит излом, который вы можете переместить в нужное место.

Мы рассмотрели блоки, отображающие вершины дерева. Помимо этого вы сами можете создавать блоки комментариев. Для этого надо нажать быструю кнопку **Comment block** (вторая снизу на рис. 9.49). Она позволит добавить на диаграмму поясняющие надписи (см. надпись сверху диаграммы на рис. 9.49). Нажатие кнопки **Comment Allude** (первая снизу) позволяет вам провести линию связи между любыми блоками, в частности, от блока комментария к одному из других блоков. Эти линии связи чисто иллюстративные и не имеют отношения к информационным связям. На рис. 9.49 подобные связи проведены от слов «Dep» и «Pers» в блоке комментария к соответствующим блокам наборов данных. Щелкнув правой кнопкой мыши на подобной связи, вы можете во всплывшем меню выбрать изображения, помещаемые в начале линии (раздел **Starts With**) и в ее конце (раздел **Ends With**).

Если у вас на рисунке блоки или связи перекрывают друг друга, вы можете менять их видимость, щелкнув правой кнопкой мыши и выбрав в контекстном меню раздел **Bring to front** (перенести наверх) или **Send to back** (перенести вниз).

Проектировщик Модулей Данных позволяет печатать сформированное дерево или диаграмму данных. Для печати выделите нужную панель и выполните команду **File | Print** или щелкните правой кнопкой мыши и выберите в контекстном меню раздел **Print**. Учтите, что печать текстов осуществляется в графическом режиме. Так что если возникают проблемы с печатью, проверьте, доступна ли в установке принтера опция **Печать текста как графики (Print Text as Graphics)**.

Мы рассмотрели Проектировщик Модулей Данных только как средство отображения и документирования информационных связей. Но в действительности возможности этого инструмента гораздо шире. Он позволяет визуальнo проектировать связи в модуле данных. Рассмотрим, как это можно делать, на следующем примере.

Пусть мы хотим спроектировать модуль данных, содержащий два набора данных базы данных **dbP**: главный, связанный с таблицей **Dep**, и вспомогательный, связанный с таблицей **Pers**. Связь должна осуществляться по полям **Dep**, содержащимся в обеих таблицах (см. раздел 9.10.1). Кроме того во втором наборе данных мы хотим ввести поле просмотра **Pro**, содержащее данные поля **Proisv** из таблицы **Dep** (см. раздел 9.10.2).

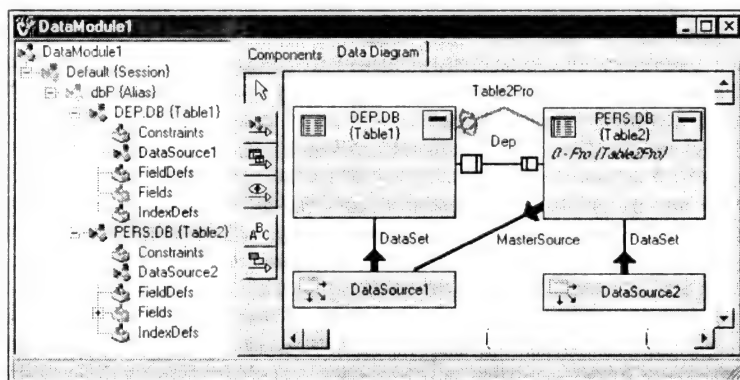
Перенесите в окно Проектировщика Модулей Данных (в панель дерева или на страницу **Components**) два компонента набора данных **Table**. Задайте в них базу данных (свойство **DatabaseName**) равным **dbP**, а имена таблиц (свойства **TableName**) задайте равным **Dep** в **Table1** и **Pers** в **Table2**. Для второго набора данных **Table2** задайте **IndexName** равным **depfio** или **IndexFieldNames** равным «Dep;Fam;Nam;Par». Это необходимо, чтобы в наборе данных **Table2** в индексе фигурировало поле **Dep**, которое потребуется для связи наборов данных друг с другом.

Теперь можно приступать к визуальнoму проектированию связей компонентов с помощью диаграммы. Откройте в Проектировщике Модулей Данных панель диаграммы и перетащите на нее из дерева вершины, связанные с **Table1** и **Table2**. Нажмите кнопку **Master Detail** (вторая сверху — см. рис. 9.50), создающую связь типа Главная таблица — Вспомогательная таблица. Щелкните мышью на блоке, отображающем **Table1**, и, не отпуская кнопку мыши, переместите курсор в блок,

отображающий **Table2**. За курсором будет тянуться линия. В момент отпускания кнопки мыши над блоком **Table2** появится диалоговое окно, рассмотренное ранее в разделе 9.10.1 (рис. 9.44). В этом окне вы должны указать, что хотите осуществить связь по полям **Dep** обеих таблиц. На диаграмме появится связь, отображающая зависимость Главная таблица — Вспомогательная таблица (рис. 9.50). Около связи будет написано имя поля (**Dep**), по которому осуществляется связь. Одновременно в дереве появится новый компонент — **DataSource1** в вершине, связанной с **Table1**. Если вы перетащите вершину **DataSource1** на страницу Data Diagram, то увидите, что новый компонент осуществляет связь наборов данных с помощью своего свойства **DataSet** и свойства **MasterSource** компонента **Table2**. В Инспекторе Объектов вы можете увидеть, что в свойствах всех компонентов установились требуемые значения.

Рис. 9.50.

Пример проектирования модуля данных с помощью диаграммы



Теперь давайте введем в набор данных **Table2** поле просмотра **Pro**. Нажмите на странице Data Diagram кнопку Lookup (третья снизу) и проведите курсором линию от блока **Table2** к блоку **Table1**. Перед вами откроется окно задания поля просмотра (см. раздел 9.10.2, рис. 9.45). Задайте в нем ту же информацию, которая показана на рис. 9.45. В результате на диаграмме в блоке **Table2** появится имя введенного поля просмотра, а между блоками **Table1** и **Table2** появится линия связи. Правда, разместится она под введенной ранее связью, отображающей соотношение главной и вспомогательной таблиц. Чтобы получить изображение, подобное рис. 9.50, можно щелкнуть правой кнопкой мыши на этой верхней связи и во всплывшем меню выбрать раздел Send to back. Тогда наверху окажется связь, определяемая полем просмотра. Далее можно курсором мыши потянуть ее за середину вверх и разместить так, как показано на рис. 9.50.

Вам осталось только добавить в модуль источник данных **DataSource2**, связанный с **Table2**. Для этого надо щелкнуть на пиктограмме **DataSource** в палитре компонентов, а затем щелкнуть в панели дерева Проектировщика Модулей Данных. Если при этом у вас будет выделена вершина **Table2**, то новый компонент автоматически свяжется с этим набором данных. Останется только перетащить его на диаграмму. Но если вы занесете источник в корневую вершину дерева, то он появится в дереве как самостоятельная вершина. Далее имеется две возможности связать ее с **Table2**. Можно в дереве перетащить эту вершину в вершину **Table2**. А можно сначала перетащить **DataSource2** на диаграмму, нажать кнопку Property (первая сверху) и провести связь от блока **DataSource2** к блоку **Table2**. Во всех вариантах результат будет одинаковым — компонент **DataSource2** свяжется с набором данных **Table2** через свое свойство **DataSet**.

Проектирование набора данных почти завершено. Осталось только выбрать в дереве или на диаграмме поочередно **Table1** и **Table2** и с помощью контекстного меню, как описывалось ранее, задать отображаемые поля наборов данных, свойства этих полей, ограничения и вычисляемые поля, если все это требуется.

Из рассмотренного примера видно, что введенный в C++Builder 5 Проектировщик Модулей Данных является мощным и удобным инструментом визуального проектирования и документирования. Особенно, конечно, он удобен в задачах, значительно более сложных, чем рассмотренная выше, с множеством наборов и источников данных и сложными связями между ними.

9.12 Пример программирования работы с базой данных

Попробуйте построить пример, использующий различные способы программирования работы с базами данных. Этот пример вы можете найти на прилагаемом к книге диске. Общий вид примера во время выполнения показан на рис. 9.51 и 9.52.

Рис. 9.51.

Приложение работы с базами данных в режиме отбора записей

Левая часть окна приложения предоставляет пользователю возможность просматривать список сотрудников по подразделениям или во всех подразделениях сразу.

Приложение содержит два компонента набора данных **Table1** и **Table2**. Набор данных **Table1** связан с таблицей **Dep** базы данных **dbP**. Это головная таблица приложения. С **Table1** связан компонент источника данных **DataSource1**. Набор данных **Table2** связан с таблицей **Pers**. Это вспомогательная таблица, связанная с **Table1**. Поэтому в **Table2** установлены свойства **MasterSource** = **DataSource1** и **MasterFields** = **Dep**. С **Table2** связан источник данных **DataSource2**, а с ним — компонент **TDBGrid**, названный **Tpers**. В этом компоненте отображается список сотрудников.

Диалоговое окно Подразделение представляет собой выпадающий список — компонент типа **TComboBox**, названный **CBdep**. В него в момент создания формы загружаются значения полей таблицы **Table1**, т.е. названия отделов. Последней строкой в список заносится строка «все отделы». При выборе пользователем этой строки в таблице **Tpers** должны отображаться сотрудники всех отделов. А при выборе пользователем в списке названия отдела в таблице **Tpers** должны отображаться только сотрудники выбранного отдела.

В качестве окна редактирования Тип подразделения, отражающего характер подразделения, можно взять компонент **DBEdit1** класса **TDBEdit**, соединенный с **Table1** через источник данных **DataSource1** и настроенный на поле **Proisrv**. При выборе пользователем подразделения в списке **CBdep** в окне **DBEdit1** должен отображаться характер этого подразделения.

Посмотрим, как все это можно обеспечить. Обработчик события **OnCreate** формы обеспечивает активизацию наборов данных и заполнение списка **CBdep**:

```
Table1->Active = true;
Table1->First();
CBdep->Clear();
while (!Table1->Eof)
{
    CBdep->Items->Add(Table1Dep->AsString);
    Table1->Next();
}
CBdep->Items->Add("все отделы");
CBdep->ItemIndex = 0;
Table1->First();
Table2->Active = true;
```

Этот код устанавливает курсор набора данных **Table1** на первую запись, очищает список **CBDep**, а затем в цикле заносит в список все значения поля **Dep** из набора данных **Table1**. Затем в список добавляется строка «все отделы», чтобы можно было в дальнейшем просматривать список сразу по всем отделам. После этого индекс списка устанавливается на первую строку, а курсор набора данных устанавливается на первую запись.

Теперь надо обеспечить управление головным набором данных **Table1** со стороны списка **CBDep**. Для этого в обработчик его события **OnChange** вставляется код:

```
if (CBdep->ItemIndex == CBdep->Items->Count-1)
{
    Table2->MasterFields = "";
    Table2->IndexFieldNames = "Fam;Nam;Par";
    DBEdit1->DataSource = NULL;
}
else
{
    Table2->MasterFields = "Dep";
    Table2->IndexFieldNames = "Dep;Fam;Nam;Par";
    Table1->FindNearest(&TVarRec(CBdep->Text), 0);
    DBEdit1->DataSource = DataSource1;
}
/* Передача фокуса таблице Tpers, иначе в ней
   не отразятся изменения */
TPers->SetFocus();
```

Если в списке выбрана последняя строка («все отделы»), связь таблиц **Table1** и **Table2** разрывается заданием пустого значения свойства **MasterFields** в таблице **Table2**. Разрывается связь окна **DBEdit1** с источником данных. Таблица **Table2** индексируется по алфавиту сотрудников. Последний оператор обработчика передает фокус компоненту **Tpers**. Этот оператор необходим, так как без него не будет видно изменение в **Tpers** сразу после изменения в списке **CBDep**.

Если в списке **CBDep** выбран один из отделов, то восстанавливается связь **Table2** и **DBEdit1** с таблицей **Table1**. Изменяется индексация вспомогательной таблицы **Table2**, чтобы в индекс входило ключевое поле **Dep**, по которому связаны таблицы. Методом **FindNearest** осуществляется поиск в **Table1** записи, соответствующей выбранному отделу. При изменении текущей записи набора данных **Table1** автоматически изменяются данные, отображаемые в **Tpers** и **DBEdit1**.

Теперь рассмотрим вспомогательные элементы левой страницы приложения. Индикатор Подразделение введен в приложение для того, чтобы при отображении полного списка сотрудников всех подразделений пользователь мог бы посмотреть, кто в каком отделе работает. Обработчик события **OnClick** индикатора имеет вид:

```
Table2Dep->Visible = CheckBox1->Checked;
```

В зависимости от состояния индикатора этот оператор делает видимым или невидимым столбец поля **Dep** в таблице.

Внизу левой части окна приложения расположен навигатор и кнопка Больше, которая делает видимой форму **Form2**, содержащую характеристику и фотографию сотрудника. Эта форма уже рассматривалась выше в разделе 9.7 (см. рис. 9.41).

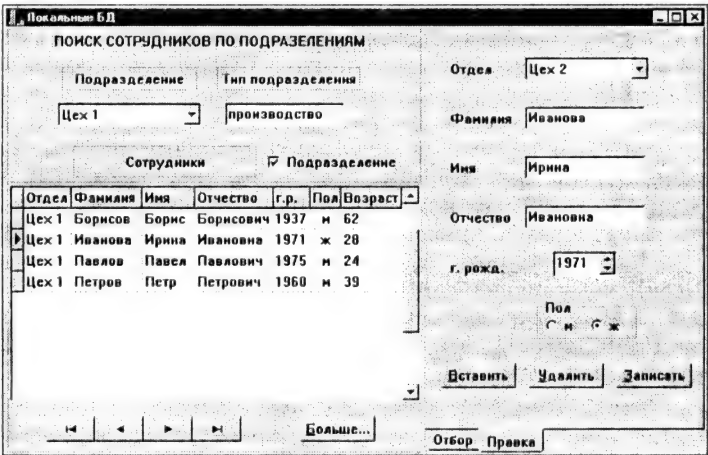
Теперь перейдем к рассмотрению правой части окна приложения. Ее занимает двухстраничный компонент **PageControl1** класса **TPageControl**. Страница Отбор представлена на рис. 9.51. Фильтрация данных в несколько другом варианте уже была рассмотрена в разделе 9.5.5, где вы и можете посмотреть, как она осуществляется. Окно **Edit1** быстрого поиска фамилии имеет обработчик события **OnChange** вида:

```
if (CBdep->Text == "все отделы")
    Table2->FindNearest (&TVarRec (Edit1->Text), 0);
else
{
    TLocateOptions SearchOptions;
    Variant locvalues[] = {CBdep->Text, Edit1->Text};
    Table2->Locate("Dep;Fam", VarArrayOf(locvalues,1),
        SearchOptions<<loPartialKey<<loCaseInsensitive);
}
```

Когда установлен режим «все отделы», поиск проводится методом **FindNearest** (это сделано просто для того, чтобы показать разные методы — можно было бы использовать и **Locate**). При этом в качестве ключа в метод передается содержимое окна **Edit1**. В остальных вариантах (при заданном отделе) поиск осуществляется методом **Locate**, в который передается два параметра — отдел, указанный в **CBdep->Text**, и содержимое **Edit1->Text**.

Рассмотрим теперь вторую страницу компонента **PageControl1** — страницу Правка, представленную на рис. 9.52. Эта страница предназначена для редактирования текущей записи. В настоящем приложении такая страница, конечно, должна открываться, только если пользователь сообщил пароль, допускающий его к редактированию базы данных.

Рис. 9.52.
Приложение работы с базами данных в режиме правки записи



На этой странице пользователь может изменить значения полей текущей записи, например, перевести сотрудника из одного отдела в другой (именно этот момент изображен на рисунке). Щелкнув на кнопке Больше, он может посмотреть и изменить характеристику сотрудника. Изменение фиксируется программой по значению свойства **Modified** компонента **DBMemo1**, расположенного на дополни-

тельной форме (см. рис. 9.41), вызываемой этой кнопкой. На той же дополнительной форме пользователь может изменить фотографию сотрудника, занеся изображение из файла с этой фотографией в буфер Clipboard, а затем щелкнув на окне фотографии **DBImage1** и скопировав файл из Clipboard быстрыми клавишами Ctrl-V. Факт того, что пользователь изменил фотографию фиксируется по событию **OnClick** окна, содержащего ее изображение. Для этого в коде предусмотрена переменная **ChangePhoto**, которая при щелчке на фотографии устанавливается в **true**.

Все окна и другие компоненты отображения данных на странице Правка — обычные компоненты, не связанные с данными. Т.е. используются, например, компоненты **Edit**, а не **DBEdit**. Это позволяет развязать процесс редактирования информации и процесс пересылки ее в базу данных. В таблице **TPers** в левой части окна приложения задано в свойстве **Options** подсвойство **dgEditing** = **false**, что не дает возможности пользователю непосредственно редактировать данные в таблице.

В компоненты страницы Правка заносятся значения полей таблицы **Table2** с помощью размещенного в обработчике события **OnAfterScroll** компонента **Table2** кода:

```
if (PageControll1->ActivePage == TabEdit)
{
    RGF->ItemIndex = 0;
    CBEDep->ItemIndex =
        CBEDep->Items->IndexOf(Table2Dep->AsString);
    EFam->Text = Table2Fam->AsString;
    ENam->Text = Table2Nam->AsString;
    EPar->Text = Table2Par->AsString;
    SEYear->Value = Table2Year_b->AsInteger;
    if (Table2Sex->AsBoolean)
        RGSex->ItemIndex = 0;
    else RGSex->ItemIndex = 1;
}
```

В этом коде следует обратить внимание на оператор

```
CBEDep->ItemIndex = CBEDep->Items->IndexOf(Table2Dep->AsString);
```

Почему он не записан проще:

```
CBEDep->Text = Table2Dep->AsString;
```

т.е. почему свойству выпадающего списка **CBEDep->Text** просто не присвоить значение поля **Dep**? Дело в том, что в **CBEDep** установлено свойство **Style** = **csDropDownList**, которое исключает возможность для пользователя редактировать текст и тем самым указать, например, неверное название подразделения. Но при этом и программно невозможно присвоить значение свойству **Text**. Поэтому приходится с помощью метода **IndexOf**, присущего свойству **CBEDep->Items** (как любой переменной типа **TStrings**), искать индекс строки, текст которой совпадает со значением поля, а затем присваивать этот индекс свойству **CBEDep->ItemIndex**.

К приведенному выше коду осуществляется обращение при событии **OnAfterScroll** компонента **Table2**, т.е. после любого перемещения по набору данных **Table2**. К этому же коду осуществляется обращение при событии **OnChange** компонента **PageControll1**, т.е. при переключении страницы. Только при переключении страницы к нему еще добавляется оператор

```
Form2->DBMemol->ReadOnly =
    ! (PageControll1->ActivePage == TabEdit);
```

который делает окно просмотра характеристики доступным или недоступным для редактирования в зависимости от того, какая страница открыта.

Кнопка Записать осуществляет пересылку результатов редактирования в базу данных. Код обработчика ее события **OnClick** имеет вид:

```

AnsiString s;
const AnsiString s1 = ",";
s="";
if (Table2Dep->AsString != CBEDEP->Text)
    s = "отдел";
if (Table2Fam->AsString != EFam->Text)
{
    if (s != "") s += s1;
    s += " фамилию";
}
...
if (Form2->DBMemol->Modified)
{
    if (s != "") s += s1;
    s += " характеристику";
}
if (ChangePhoto)
{
    if (s != "") s += s1;
    s += " фотографию";
}
if (s != "")
if (Application->MessageBox(
    ("Действительно хотите изменить "+s+"?").c_str(),
    "Подтвердите занесение в базу данных",
    MB_YESNO + MB_ICONQUESTION) == IDYES)
{
    Table2->Edit();
    Table2Dep->AsString = CBEDEP->Text;
    Table2Fam->AsString = EFam->Text;
    Table2Nam->AsString = ENam->Text;
    Table2Par->AsString = EPar->Text;
    Table2Year b->AsInteger = SEYear->Value;
    Table2Sex->AsBoolean = (RGSex->ItemIndex == 0);
    CanPost = true;
    Table2->Post();
    CanPost = false;
    Form2->DBMemol->Modified = false;
    ChangePhoto=false;
};

```

Первые операторы этого кода, данные с некоторым сокращением, формируют запрос пользователю типа «Действительно хотите изменить отдел, характеристику?», в котором указано то, что пользователь изменил. При положительном ответе пользователя на этот запрос осуществляется пересылка данных из окон редактирования в поля и выполняется метод **Post**. В заключение сбрасывается свойство **Modified** компонента отображения характеристики и переменная **ChangePhoto**, фиксирующая изменения фотографии.

Перед вызовом метода **Post** и после этого вызова в коде вы можете видеть операторы, изменяющие переменную **CanPost**. Что это за переменная и зачем она введена в приложение? Дело в том, что надо предотвратить несанкционированную запись модифицированной характеристики через связанное с данными окно **DBMemol** на форме **Form2** или несанкционированную запись измененной фотографии через связанное с данными окно типа **TDBImage** на форме **Form2** при случайном или намеренном перемещении пользователя по набору данных. Данные должны изменяться только после щелчка пользователя на кнопке Записать и положительного ответа на запрос программы. Поэтому в обработчик события **BeforePost** компонента **Table2** введен оператор

```
if (! CanPost)
{
    DataSet->Cancel();
    Abort();
}
```

Он прерывает запись данных, если переменная **CanPost** = **false**. Т.е. при перемещениях по таблице запись производиться не будет. А при щелчке на кнопке Записать перед вызовом **Post** переменная **CanPost** делается равной **true** и в этом случае запись данных производится.

Реализация кнопок Вставить и Удалить, которые позволяют соответственно вставить новую запись или удалить текущую запись, проблем не вызывает. В обработчике события **OnClick** кнопки Вставить выполняется метод **Table2->Insert()**, после чего в окна редактирования передаются значения полей по умолчанию. После их заполнения пользователь должен щелкнуть на кнопке Записать и внесенные данные будут занесены в таблицу **Pers**.

Но для того, чтобы можно было вставить новую запись, необходимо выполнить одно условие: в набор данных **Table2** не должно передаваться поле **Num**. Это поле в базе данных задано как автоматически нарастающее и обязательно присутствующее в каждой записи. Если вы не включили это поле в набор данных **Table2**, то база данных сама даст ему уникальное значение. Если же вы включили это поле в набор данных **Table2**, то при попытке выполнить метод **Post** вы получите сообщение о том, что значение этого поля не задано. А задать вы его не сможете, так как оно автоматически изменяемое. Поэтому вам не удастся включить вставленную новую запись в базу данных.

В обработчик события **OnClick** кнопки Удалить вставляется оператор

```
if (Application->MessageBox(
    "Действительно хотите удалить запись?",
    "Подтвердите удаление записи",
    MB_YESNO + MB_ICONEXCLAMATION) == IDYES)
    Table2->Delete();
```

который после положительного ответа пользователя на вопрос программы удаляет текущую запись.

Мы рассмотрели один из возможных примеров работы с базой данных. Подробнее вы можете посмотреть этот пример на диске, прилагаемом к книге. Пользуясь сведениями о программировании работы с базами данных, изложенными ранее, вы, наверное, сможете сами спроектировать и более сложное, и более полезное для вас приложение.

Глава 10

Создание приложений для работы с базами данных в сети

10.1 Основы языка SQL и его использование в приложениях

10.1.1 Общие сведения

Язык SQL (Structured Query Language — язык структурированных запросов) был создан Microsoft в конце 70-ых годов и получил через некоторое время широкое распространение. Он позволяет формировать весьма сложные запросы к базам данных. *Запрос* — это вопрос к базе данных, возвращающий запись или множество записей, удовлетворяющих вопросу.

К сожалению, SQL в настоящее время недостаточно стандартизован. Существует стандарт SQL ANSI, но существует и множество диалектов, с которыми работают различные системы. Например, Sybase SQL Server и Microsoft SQL используют синтаксис, существенно отличающийся от стандарта ANSI. InterBase, Oracle и многие другие серверы в основном придерживаются стандарта ANSI, но каждый разработчик вносит в него и свои усовершенствования. Ниже изложение будет основываться на диалекте, принятом в локальном сервере InterBase, с которым мы позднее познакомимся. Впрочем, поскольку речь будет идти только об основных операторах языка, расхождение в их синтаксисе между различными диалектами невелико. А чтобы действительно изучить SQL, надо обратиться к документации той системы, с которой вы работаете.

C++Builder позволяет приложению при помощи запросов SQL использовать данные:

- Таблиц PARADOX и dBase — используется синтаксис локального SQL.
- Локального сервера InterBase — полностью поддерживается соответствующий синтаксис.
- Удаленных серверов SQL через драйверы SQL Links.

Общие правила синтаксиса SQL очень просты. Язык SQL не чувствителен к регистру, так что, например, рассмотренный ниже оператор **Select** можно писать и **SELECT**, и **Select**, и **select**. Если используется программа из нескольких операторов SQL, то в конце каждого оператора ставится точка с запятой «;». Впрочем, если вы используете всего один оператор, то точка с запятой в конце не обязательна. Комментарий может писаться и в стиле C: /*<комментарий>*/, а в некоторых системах и в стиле Pascal: {<комментарий>}. Вот, собственно, и все правила.

В рамках данной книги невозможно рассмотреть все конструкции языка SQL. Мы остановимся только на самых распространенных. Более подробные сведения вы можете найти в книге [8].

10.1.2 Оператор выбора Select

10.1.2.1 Отбор записей из таблицы

В этом разделе мы познакомимся с наиболее часто используемым оператором SQL — оператором выбора **Select**. Этот оператор возвращает одно или множество значений, которые могут представлять собой значения указанных полей записей, удовлетворяющих указанному условию и упорядоченных по заданному критерию.

Хотя мы еще не рассматривали компонент C++Builder **Query**, специально предназначенный для выполнения запросов SQL, но при знакомстве с оператором **Select** полезно сразу пробовать записывать его в различных вариантах и смотреть получающиеся результаты. Поэтому откройте новое приложение C++Builder, перенесите на форму компонент **Query** со страницы библиотеки Data Access и установите его свойство **DatabaseName** равным **dbP** — базе данных Paradox, с которой мы имели дело в главе 9. В некоторых случаях полезно изменить этот псевдоним на **ib** — аналогичную базу данных InterBase. Обе базы данных содержатся на прилагаемом к книге диске. Можете списать их оттуда и установить соответствующие псевдонимы (см. раздел 9.3).

Поместите на форму компонент **DataSource** и в его свойстве **DataSet** задайте **Query1**. Поместите также на форму компонент **DBGrid** и в его свойстве **DataSource** задайте **DataSource1**.

Теперь ваше тестовое приложение для экспериментов с языком SQL готово. Операторы SQL вы можете писать в свойстве **SQL** компонента **Query1**, а чтобы увидеть результаты выполнения написанного оператора, вам надо будет устанавливать значение свойства **Active** компонента **Query1** в **true**. Это надо будет делать после записи каждого нового оператора.

Теперь начнем рассмотрение оператора **Select**. Одна из форм этого оператора имеет синтаксис:

```
SELECT <список имен полей> FROM <таблица>
WHERE <условие отбора> ORDER BY <список имен полей>;
```

Элементы оператора **WHERE** и **ORDER BY** не являются обязательными. Элемент **WHERE** определяет условие отбора записей: отбираются только те, в которых условие выполняется. Элемент **ORDER BY** определяет упорядочивание возвращаемых записей.

<таблица> — это не компонент **Table**, а именно та таблица базы данных, из которой осуществляется отбор, например, **Pers**.

Начнем подробное рассмотрение данного оператора со списка полей после ключевого слова **Select**, содержащего имена тех полей таблицы, которые будут возвращены. Имена разделяются запятыми. Например, оператор

```
SELECT Fam, Nam, Par, Year_b FROM Pers
```

указывает, что следует вернуть поля **Fam**, **Nam**, **Par** и **Year_b** из таблицы **Pers**. Запишите его в свойстве **SQL** компонента **Query1**, установите значение свойства **Active** компонента **Query1** в **true** и посмотрите результаты.

Если указать вместо списка полей символ ***** — это будет означать, что требуется вернуть все поля. Например, оператор

```
SELECT * FROM Pers
```

означает выбор всех полей.

В списке могут быть не только сами поля, но и любые выражения от них с арифметическими операциями **+**, **-**, *****, **/**. После выражения может записываться псевдоним выражения в форме: **AS <псевдоним>**. В качестве псевдонима может фигурировать любой идентификатор, на который потом можно будет при необходимости ссылаться. Указанный псевдоним будет при отображении результатов фигурировать в заголовке таблицы.

Такими простыми средствами создаются аналоги вычисляемых полей, которые при использовании компонента **Table** требовали гораздо больших усилий и написания обработчика события **OnCalcFields**. Впрочем, в обработчике можно было написать сколь угодно сложную программу вычисления, что в данном случае затруднительно. Правда, как мы увидим позже, компонент **Query**, работающий в C++Builder с SQL, позволяет создавать вычисляемые поля и с помощью обработчика события **OnCalcFields**.

Приведем пример использования выражения:

```
SELECT Fam, Nam, (2000-Year_b) AS Age FROM Pers
```

Этот оператор создает поле **Age**, вычисляемое по формуле **(2000-Year_b)**.

Теперь рассмотрим форму представления условия отбора, задаваемого после ключевого слова **WHERE**. Это условие определяет критерий, по которому отбираются записи. Оператор **Select** отбирает только те записи, в которых заданное условие истинно. Условие может включать имена полей (кроме вычисляемых), константы, логические выражения, содержащие арифметические операции, логические операции **and**, **or**, **not** и операции отношения:

=	равно
>	больше
>=	больше или равно
<	меньше
<=	меньше или равно
!=	не равно
Like	наличие заданной последовательности символов
between ... and	диапазон значений
in	соответствие элементу множества

Первые шесть операций очевидны. Например, оператор

```
SELECT Fam FROM Pers WHERE Sex=false and Year_b > 1960
```

отберет записи, относящиеся к женщинам, родившимся после 1960 года.

Операция **Like** имеет синтаксис:

```
<поле> LIKE '<последовательность символов>'
```

Эта операция применима к полям типа строк и возвращает **true**, если в строке встретился фрагмент, заданный в операции как **<последовательность символов>**. Заданным символам может предшествовать и их может завершать символ процента **%**, который означает — любое количество любых символов. Если символ процента не указан, то заданная последовательность символов должна соответствовать только целому слову. Например, условие

```
Fam LIKE 'A%'
```

означает, что будут отобраны все записи, начинающиеся с заглавной русской буквы «А» (операция **Like** различает строчные и прописные символы). Условию

```
Fam LIKE 'Иванов%'
```

будут удовлетворять фамилии «Иванов» и «Иванова», а условию

```
Fam LIKE '%ван%'
```

кроме этих фамилий будет удовлетворять, например, фамилия «Иванников».

Операция **between ... and** имеет синтаксис:

<поле> between <значение> and <значение>

и задает для указанного поля диапазон отбираемых значений. Например, оператор

```
SELECT Fam, Year_b FROM Pers WHERE Year_b BETWEEN 1960 AND 1970
```

отберет записи сотрудников в заданном диапазоне возраста (включая граничные значения 1960 и 1970).

Операция **In** имеет синтаксис:

<поле> in (<множество>)

и отбирает записи, в которых значение указанного поля является одним из элементов указанного множества. Например, оператор

```
SELECT Fam, Year_b FROM Pers WHERE Fam IN('Иванов', 'Петров', 'Сидоров')
```

отберет записи сотрудников с заданными фамилиями, а оператор

```
SELECT Fam, Year_b FROM pers WHERE Year_b IN(1950,1960)
```

отберет записи сотрудников указанных годов рождения.

Элемент оператора **Select**, начинающийся с ключевых слов **ORDER BY**, определяет упорядочивание (сортировку) записей. После этих ключевых слов следует список полей, определяющих сортировку. Можно указывать только поля, фигурирующие в списке отобранных (в списке после ключевого слова **SELECT**). Причем эти поля могут быть и вычисляемыми.

Если в списке сортировки указано только одно поле, то сортировка производится по умолчанию в порядке нарастания значений этого поля. Например, оператор

```
SELECT Dep, Fam, Year_b FROM Pers ORDER BY Year_b
```

задает упорядочивание возвращаемых значений по нарастанию года рождения. Если желательно располагать результаты по убыванию значений, то после имени поля добавляется ключевое слово **DESC**:

```
SELECT Dep, Fam, Year_b FROM Pers ORDER BY Year_b DESC
```

Если в списке после **ORDER BY** перечисляется несколько полей, то первое из них — главное и сортировка проводится прежде всего по значениям этого поля. Записи, имеющие одинаковое значение первого поля упорядочиваются по значениям второго поля и т.д. Например, оператор

```
SELECT Dep, Fam, Year_b FROM Pers ORDER BY Dep, Fam
```

сортирует записи прежде всего по отделам (значениям поля **Dep**), а внутри каждого отдела — по алфавиту. Оператор

```
SELECT Dep, Fam, Year_b, Sex FROM Pers ORDER BY Dep, Sex, Fam
```

сортирует записи по отделам, полу и алфавиту.

10.1.2.2 Совокупные характеристики

Оператор **Select** позволяет возвращать не только множество значений полей, но и некоторые совокупные (агрегированные) характеристики, подсчитанные по всем или по указанным записям таблицы. Одна из функций, возвращающих такие совокупные характеристики, **count(<условие>)** — количество записей в таблице, удовлетворяющих заданным условиям. Например, оператор

```
SELECT count(*) FROM Pers
```

подсчитает полное количество записей в таблице **Pers**. А оператор

```
SELECT count(*) FROM Pers WHERE Dep='Цех 1'
```

выдаст число записей сотрудников цеха 1.

Оператор, использующий ключевое слово **DISTINCT** (уникальный), выдаст число неповторяющихся значений в указанном поле. Например, оператор

```
SELECT count(DISTINCT Dep) FROM Pers
```

вернет число различных подразделений, упомянутых в поле **Dep** таблицы **Pers**.

Функции **min(<поле>)**, **max(<поле>)**, **avg(<поле>)**, **sum(<поле>)** возвращают соответственно минимальное, максимальное, среднее и суммарное значения указанного поля. Например, оператор

```
SELECT min(Year_b), max(Year_b), avg(Year_b) FROM Pers
```

вернет минимальное, максимальное и среднее значение года рождения, а оператор

```
SELECT min(2000-Year_b), max(2000-Year_b), avg(2000-Year_b)  
FROM Pers WHERE Dep='Бухгалтерия'
```

выдаст вам аналогичные данные, но относящиеся к возрасту сотрудников бухгалтерии.

В операторе **Select** вы можете указывать не только суммарные характеристики, но и любые выражения от них. Например, оператор

```
SELECT 1999-(min(Year_b)+max(Year_b))/2 FROM Pers  
WHERE Dep='Бухгалтерия'
```

выдаст моду (среднее между максимальным и минимальным значениями) возраста сотрудников бухгалтерии. Здесь надо обратить внимание на то, что оператор вернет округленное до целого значение моду, поскольку в выражении использованы только целые числа и поэтому осуществляется целочисленное деление. Если же вы в том же операторе замените делитель «2» на «2.», т.е. укажете его как действительное значение, то и результат будет представлен действительным числом.

При использовании суммарных характеристик надо учитывать, что в списке возвращаемых значений после ключевого слова **SELECT** могут фигурировать или поля (в том числе вычисляемые), или совокупные характеристики, но не могут фигурировать и те, и другие (без указания на группирование данных, о чем будет сказано ниже). Это очевидно, так как оператор может возвращать или множество значений полей записей, или суммарные характеристики по таблице, но не может возвращать эти несовместимые друг с другом данные. Поэтому нельзя, например, записать оператор

```
SELECT Fam, max(Year_b) FROM Pers
```

в котором мы пытаемся определить фамилию самого молодого сотрудника. Впрочем, эту задачу можно решить с помощью вложенных запросов, которые рассмотрены ниже.

Смешение в одном операторе полей и совокупных характеристик возможно, если использовать группировку записей, задаваемую ключевыми словами **GROUP BY**. После этих ключевых слов перечисляются все поля, входящие в список **SELECT**. В этом случае смысл совокупных характеристик изменяется: они проводят вычисления не по всем записям таблицы, а по тем, которые соответствуют одинаковым значениям указанных полей. Например, оператор

```
SELECT Dep, count(*) FROM Pers GROUP BY Dep
```

вернет таблицу, в которой будет 2 столбца: столбец с названиями отделов, и столбец, в котором будет отображено число сотрудников в каждом отделе:

При группировании записей с помощью **GROUP BY** можно вводить условия отбора записей с помощью ключевого слова **HAVING**. Например, если переписать приведенный выше оператор следующим образом:

```
SELECT Dep, count(*) FROM Pers GROUP BY Dep HAVING Dep <> 'Бухгалтерия'
```

то в таблице будут строки, относящиеся ко всем отделам, кроме бухгалтерии.

Впрочем, не все базы данных поддерживают синтаксис **HAVING**. При использовании базы данных **dbP (Paradox)** компонент **Query** не работает с **HAVING**. Для базы данных **InterBase** никаких проблем с **HAVING** не возникает.

10.1.2.3 Вложенные запросы

Результаты, возвращаемые оператором **Select**, можно использовать в другом операторе **Select**. Причем это относится и к операторам, возвращающим совокупные характеристики, и к операторам, возвращающим множество значений. Например, в предыдущем разделе нам не удалось узнать фамилию самого молодого сотрудника. Теперь это можно сделать с помощью вложенных запросов:

```
SELECT Fam, Year_b FROM Pers
WHERE Year_b=(SELECT max(Year_b) FROM Pers)
```

В этом операторе второй вложенный оператор **SELECT max(Year_b) FROM Pers** возвращает максимальный год рождения, который используется в элементе **WHERE** основного оператора **Select** для поиска сотрудника (или сотрудников), чей год рождения совпадает с максимальным.

Вложенные запросы могут обращаться к разным таблицам. Пусть, например, мы имеем две аналогичных по структуре таблицы **Pers** и **Pers1**, относящиеся к разным организациям, и хотим в таблице **Pers** найти всех однофамильцев сотрудников другой организации. Чтобы проверить работу с несколькими таблицами, вы можете открыть в **Database Desktop** свою таблицу **Pers**, выполнить команду **Table | Restructure**, ничего не меняя в структуре, щелкнуть на кнопке **Save as** и сохранить ту же таблицу в том же каталоге, где была исходная таблица **Pers**, но под новым именем **Pers1**. Если хотите, можно в ней внести записи, отличные от таблицы **Pers**, чтобы эти таблицы чем-то различались. Впрочем, можно все это и не делать, заменив в приведенных ниже примерах таблицу **Pers1** таблицей **Pers**, т.е. работая с одной таблицей. Смысл операторов все равно будет понятен.

Итак, вернемся к задаче определения всех однофамильцев в этих двух таблицах. Это можно сделать оператором

```
SELECT * FROM Pers WHERE Fam IN (SELECT Fam FROM Pers1)
```

Вложенный оператор **Select Fam from Pers1** возвращает множество фамилий из таблицы **Pers1**, а конструкция **WHERE** основного оператора **Select** отбирает из фамилий в таблице **Pers** те, которые имеются в множестве фамилий из **Pers1**.

При работе в условии **WHERE** с множествами записей можно использовать ключевые слова: **All** и **Any**. **All** означает, что условие выполняется для всех записей, а **Any** — хотя бы для одной записи. Например, оператор

```
SELECT * FROM Pers WHERE Year_b >= ALL (SELECT Year_b FROM Pers1)
```

ищет сотрудников в **Pers**, которые не старше любого сотрудника в **Pers1**. Кстати, если в этом операторе заменить **Pers1** на **Pers**, то получим список самых молодых сотрудников организации, который мы получали ранее другим способом. А оператор

```
SELECT * FROM Pers WHERE Year_b > ANY (SELECT Year_b FROM Pers1)
```

ищет сотрудников в **Pers**, которые моложе хотя бы одного сотрудника в **Pers1**.

10.1.2.4 Объединения таблиц

В запросе можно объединить данные двух или более таблиц. Пусть, например, вы хотите получить список сотрудников всех производственных подразделений. В таблице **Pers** мы имеем список сотрудников с указанием в поле **Dep** подразделений, в которых они работают. А в таблице **Dep** мы имеем список всех подразделений в поле **Dep** и характеристику каждого подразделения в поле **Prosv** (**true**, если подразделение производственное). Тогда получить список сотрудников всех производственных подразделений можно оператором:

```
SELECT Pers.* FROM Pers, Dep
WHERE (Pers.Dep=Dep.Dep)AND (Dep.Proisv=true)
```

В нем мы обращаемся сразу к двум таблицам Pers и Dep, которые перечислены после ключевого слова **FROM**. Поэтому каждое имя поля предворяется ссылкой на таблицу, к которой оно относится. Впрочем, это надо делать только для полей, имя которых повторяется в разных таблицах (поле **Dep**). Перед полем **Proisv** ссылку на таблицу можно опустить. В конструкции **WHERE** условие **Pers.Dep=Dep.Dep** ищет запись в таблице Dep, в которой поле **Dep** совпадает с полем **Dep** текущей записи таблицы Pers. А условие **Dep.Proisv=true** отбирает те записи, в которых в таблице Dep найденному подразделению соответствует поле **Proisv = true**.

В операторах, работающих с несколькими таблицами, обычно каждой таблице дается псевдоним, сокращающий ссылки на таблицы, а иногда придающий им некоторый смысл, вытекающий из данного применения. Псевдоним таблицы может записываться в списке таблиц после слова **FROM**, отделяясь от имени таблицы пробелом. Например, приведенный выше оператор может быть переписан следующим образом:

```
SELECT P.* FROM Pers P, Dep D
WHERE (P.Dep=D.Dep)AND (D.Proisv=true)
```

В этом примере таблице Pers дан псевдоним **P**, а таблице Dep — **D**. Конечно, эти псевдонимы действуют только в данном операторе и не имеют никакого отношения к псевдонимам баз данных, которые мы постоянно используем.

Возможно самообъединение таблицы. В этом случае одной таблице даются два псевдонима. Пусть, например, мы хотим найти всех ровесников в организации. Это можно сделать оператором

```
SELECT p1.fam, p2.fam, p1.year_b FROM Pers p1, Pers p2
WHERE (p1.year_b = p2.year_b) AND (p1.fam != p2.fam)
```

В этом примере для таблицы Pers мы ввели два псевдонима: **p1** и **p2**. В конструкции **WHERE** мы ищем в этих якобы разных таблицах записи с одинаковым годом рождения. Второе условие **p1.fam != p2.fam** нужно, чтобы сотрудник не отождествлялся в результатах как ровесник сам себя. Правда, приведенный оператор выдает в результате по две записи на каждую пару ровесников, сначала, например, «Николаев — Андреев», а потом «Андреев — Николаев». Чтобы исключить такое дублирование можно добавить еще одно условие — **p1.Fam < p2.Fam**:

```
SELECT p1.fam, p2.fam, p1.year_b FROM Pers p1, Pers p2
WHERE (p1.year_b = p2.year_b) AND (p1.fam != p2.fam)
and (p1.Fam < p2.Fam)
```

Дополнительное условие упорядочивает появление фамилий в **p1** и **p2** и исключает дублирование результатов.

До сих пор мы рассматривали объединения, основанные на однозначном соответствии записей двух таблиц, когда каждой записи в первой таблице находилась соответствующая ей запись во второй таблице. Возможны и другие виды объединений, которые выдают записи независимо от того, есть ли соответствующее поле во второй таблице. Это *внешние объединения* (*outer join*). Их три типа: левое, правое и полное. Левое объединение (обозначается ключевыми словами **LEFT OUTER JOIN ... ON**) включает в результат все записи первой таблицы, даже те, для которых не имеется соответствия во второй. Правое объединение (обозначается ключевыми словами **RIGHT OUTER JOIN ... ON**) включает в результат все записи второй таблицы, даже если им нет соответствия в записях первой. Полное объединение (обозначается ключевыми словами **FULL OUTER JOIN ... ON**) включает в результат объединение записей обеих таблиц, независимо от их соответствия.

Пусть, например, у вас есть таблица сотрудников некоей компании Pers и есть таблица Chef, в которой занесены данные на членов совета директоров этой компании. В число членов совета входят и сотрудники компании, и посторонние лица. Для определенности положим, что в таблице Pers имеются записи на сотрудников

«Иванов» и «Петров», причем Петров является членом совета, а Иванов — нет. В таблице Chef имеются записи на членов совета «Петров» и «Сидоров», причем Сидоров — не сотрудник компании. Тогда оператор

```
SELECT * FROM Pers LEFT OUTER JOIN Chef ON Pers.Fam = Chef.Fam
```

выдаст результат вида:

Поля таблицы Pers		Поля таблицы Chef	
Иванов		
Петров	Петров

Оператор задал левое объединение таблицы Pers (она указана после ключевого слова **FROM**) с таблицей Chef (она указана после ключевых слов **LEFT OUTER JOIN**). Условие объединения указано после ключевого слова **ON** и заключается в совпадении фамилий.

Как показано, результат включает все поля и таблицы Pers, и таблицы Chef. Число строк соответствует числу записей таблицы Pers. В строках, относящихся к записям, для которых в Chef не нашлось соответствие, поля таблицы Chef остаются пустые.

Оператор правого объединения

```
SELECT * FROM Pers RIGHT OUTER JOIN Chef ON Pers.Fam = Chef.Fam
```

выдаст результат вида:

Поля таблицы Pers		Поля таблицы Chef	
Петров	Петров
		Сидоров

Число строк соответствует числу записей таблицы Chef. В строках, относящихся к записям, для которых в Pers не нашлось соответствие, поля таблицы Pers остаются пустые.

Оператор полного объединения

```
SELECT * FROM Pers FULL OUTER JOIN Chef ON Pers.Fam = Chef.Fam
```

выдаст результат вида:

Поля таблицы Pers		Поля таблицы Chef	
Иванов		
Петров	Петров
		Сидоров

В нем к строкам, относящимся к таблице Pers, добавлены строки, относящиеся к таблице Chef, для которых не нашлось соответствия в таблице Pers.

10.1.3 Операции с записями

В этом и нескольких последующих разделах вы уже не сможете проверять операторы SQL с помощью вашего тестового приложения. Или обойдитесь пока без проверки, или посмотрите сначала раздел 10.3.3 и проверяйте операторы с помощью программы WISQL.

Вставка новой записи в таблицу осуществляется оператором **Insert**, который может иметь вид:

```
INSERT INTO <имя таблицы> (<список полей>) VALUES (<список значений>)
```

В списке перечисляются только те поля, значения которых известны. Остальные могут опускаться. Для пропущенных полей значения берутся по умолчанию (если значения по умолчанию заданы) или поля остаются пустыми.

Например:

```
INSERT INTO Pers (Fam, Nam, Par, Sex)
VALUES ('Иванов', 'Андрей', 'Андреевич', true)
```

В этом примере не указан год рождения. Он подставится по умолчанию и в дальнейшем может быть уточнен.

Другая форма оператора **Insert** использует множество значений, возвращаемых оператором **Select**. Этот оператор может выбирать записи из какой-то другой таблицы и вставлять их в данную. Синтаксис этой формы **Insert** :

```
INSERT INTO <имя таблицы> <оператор Select>
```

Пусть, например, вы создали таблицу **Old_Pers** пожилых людей вашей организации и хотите заполнить ее соответствующими записями из таблицы **Pers**. Это можно сделать одним оператором:

```
INSERT INTO Old_Pers SELECT * FROM Pers WHERE Year_b < 1939
```

Таблица **Old_Pers** сразу заполнится множеством соответствующих записей из **Pers**.

Приведенную форму оператора **Insert** можно использовать для копирования всех данных одной таблицы в другую, причем эти таблицы могут быть созданы разными СУБД.

Редактирование записей осуществляется оператором **Update**:

```
UPDATE <имя таблицы> SET <список вида <поле>=<выражение>>
WHERE <условие>
```

Наличие в этом операторе условия позволяет редактировать не только одну запись, но сразу множество их. Например, если при очередной реорганизации предприятия решили слить «Цех 1» и «Цех 2» в один «Цех 1», то исправление всех записей в таблице можно сделать одним оператором:

```
UPDATE Pers SET Dep = 'Цех 1' WHERE Dep = 'Цех 2'
```

Удаление записей осуществляется оператором **Delete**:

```
DELETE FROM <имя таблицы> WHERE <условие>
```

Наличие в операторе условия позволяет удалять не только одну, но сразу множество записей. Например, если при реорганизации предприятия подразделение «Цех 1» ликвидировали и всех его сотрудников уволили из штата данной организации, то удалить из таблицы все соответствующие записи можно оператором:

```
DELETE FROM Pers WHERE Dep = 'Цех 1'
```

10.1.4 Операции с таблицами

Создание новой таблицы осуществляется оператором **Create Table**:

```
CREATE TABLE <имя таблицы> (<список вида <имя поля> <тип>(<размер>)>)
```

Размер указывается только для полей строковых и некоторых других типов. После объявления некоторых полей могут включаться слова **PRIMARY KEY**, что указывает на то, что данное поле входит в первичный ключ. Кроме того после объявления некоторых полей можно вставлять слова **NOT NULL**, означающие, что значение этого поля обязательно должно быть задано в каждой записи. Например:


```
CREATE TABLE Person (
    Fam char(15) NOT NULL PRIMARY KEY,
    Nam char(15) NOT NULL PRIMARY KEY,
    Par char(15) NOT NULL PRIMARY KEY,
    Year_b integer
)
```

Приведенная форма оператора **Create Table** — простейшая. Более сложные формы позволяют задавать в таблице вычисляемые поля, значения по умолчанию, ограничения значений, связывать разные таблицы по ключам и многое другое. Подробнее это все рассмотрено в книге [8].

Удаление таблицы осуществляется оператором **Drop Table**:

```
DROP TABLE <имя таблицы>
```

Надо учесть, что удаление таблицы в корне отличается от удаления в ней всех записей. При удалении даже всех записей сама таблица (ее структура) остается, а оператор **Drop Table** полностью уничтожает таблицу.

Модификация структуры существующей таблицы осуществляется оператором **Alter Table**:

```
ALTER TABLE <имя таблицы> <действие> <имя поля> <тип данных> ...
```

В этом операторе **<действие>** может принимать значения **ADD** — добавить новое поле, или **DROP** — удалить существующее поле. Если поле добавляется, то для него надо указывать **<тип данных>**. Если поле удаляется, то тип данных не указывается. Приведем пример оператора модификации структуры:

```
ALTER TABLE Pers DROP Year_b, ADD Age integer
```

10.1.5 Операции с индексами

Индексы существенно ускоряют процесс поиска и упорядочивания записей таблицы. Если в операторе **Select** содержится элемент упорядочивания **ORDER BY** и перечисляемые поля совпадают с определенными в индексе, упорядочивание будет использовать этот индекс и произойдет с малыми затратами времени. В противном случае индекс использоваться не будет и упорядочивание потребует большего времени.

Создание нового индекса осуществляется оператором **Create Index**:

```
CREATE INDEX <имя индекса> ON <имя таблицы > <список полей>
```

Например:

```
CREATE INDEX depyear ON Pers Dep, Year_b
```

Удаление существующего индекса осуществляется оператором **Drop Index**:

```
DROP INDEX <имя таблицы >.<имя индекса>
```

Например:

```
DROP Index Pers.depyear
```

Если таблица многократно изменяется и в нее вносится много новых записей, индексы могут оказаться разбалансированы и их эффективность при выполнении запросов уменьшается. В этом случае полезно проводить повторное создание и балансировку индекса последовательным применением операторов деактивации и активации:

```
ALTER INDEX <имя индекса> DEACTIVATE
ALTER INDEX <имя индекса> ACTIVATE
```

10.1.6 Компонент Query

10.1.6.1 Общие сведения

Мы рассмотрели основные операторы SQL. Теперь посмотрим, как этот язык можно использовать в приложениях. Для этого существует компонент набора данных класса **TQuery**. Он имеет большинство свойств и методов, совпадающих с **Table**, и компонент **Query** может во многих случаях включаться в приложения вместо **Table**. Дополнительные преимущества **Query** — возможность формировать запросы на языке SQL.

Рассмотрим коротко сравнительные характеристики **Table** и **Query**. При работе с локальными базами данных чаще используется **Table**. С его помощью проще не только просматривать таблицу базы данных, но и модифицировать записи, удалять их, вставлять новые записи. Однако, при работе с серверными базами данных компонент **Table** становится мало эффективным. В этом случае он создает на компьютере пользователя временную копию серверной базы данных и работает с этой копией. Естественно, что подобная процедура требует больших ресурсов и существенно загружает сеть.

Этот недостаток отсутствует в компоненте **Query**. Если запрос SQL сводится к просмотру таблицы (запрос **Select**), то результаты этого запроса (а не сама исходная таблица) помещаются во временном файле на компьютере пользователя. Правда, в отличие от набора данных, создаваемого **Table**, это таблица только для чтения и не допускает каких-то изменений. Впрочем, это ограничение можно обойти, и в дальнейшем будет показано, как это можно делать. Если же запрос SQL связан с какими-то изменениями содержания таблицы, то никаких временных таблиц не создается. BDE передает запрос на сервер, там он обрабатывается и в приложение возвращается информация о том, успешно ли завершена соответствующая операция. Благодаря такой организации работы эффективность **Query** при работе в сети становится много выше, чем эффективность **Table**. К тому же язык SQL позволяет формулировать сложные запросы, которые не всегда можно реализовать в **Table**.

С другой стороны при работе с локальными базами данных эффективность **Query** заметно ниже эффективности **Table**. Замедление вычислений получается весьма ощутимым.

Исходя из этого краткого обзора возможностей **Table** и **Query**, можно заключить, что в серверных приложениях обычно целесообразнее использовать компонент **Query**, а при работе с локальными базами данных — компонент **Table**.

Чтобы ознакомиться с **Query**, откройте в C++Builder новое приложение и поместите на форму компоненты **Query**, **DataSource**, **DBGrid**. В свойстве **DataSet** компонента **DataSource1** задайте **Query1**, а в свойстве **DataSource** компонента **DBGrid1** задайте **DataSource1**. Таким образом, мы создали обычную цепочку: набор данных (**Query1**), источник данных (**DataSource1**), компонент визуализации и управления данными (**DBGrid1**). А теперь займемся интересующим нас компонентом **Query**.

Основное свойство компонента **Query** — **SQL**, имеющее тип **TStrings**. Это список строк, содержащих запросы SQL. В процессе проектирования приложения обычно необходимо, как будет показано ниже, сформировать в этом свойстве некоторый предварительный запрос SQL, который показал бы, с какой таблицей или таблицами будет проводиться работа. Но далее во время выполнения приложения свойство **SQL** может формироваться программно методами, обычными для класса **TStrings**: **Clear** — очистка, **Add** — добавление строки и т.д.

Настройку компонента **Query** в процессе проектирования можно производить вручную, как это делается со всеми компонентами, или с помощью специального Визуального Построителя Запросов. Его вызов производится щелчком правой кнопки мыши на компоненте **Query** и выбором из всплывающего меню раздела SQL Builder. Останавливаться на работе с Визуальным Построителем Запросов мы не бу-

дем. Ознакомьтесь с ним самостоятельно, пользуясь встроенной справкой C++Builder. Работа с ним рассмотрена также в книге [8]. Следует сказать, что возможности визуального построителя запросов в целом не очень велики, так что обычно ручная настройка предпочтительнее.

Прежде, чем приступать к ручной настройке, в свойстве **DataBaseName** компонента **Query** надо задать, как это делается и для компонентов **Table**, базу данных, с которой будет осуществляться связь. База данных задается выбором из выпадающего списка псевдонимов, или указанием полного пути к каталогу или файлу (в зависимости от используемой СУБД).

Предупреждение

Не устанавливайте свойство **DataSource** — как вы увидите позднее, это свойство имеет отношение к приложениям с несколькими связанными таблицами и в других случаях не устанавливается.

Свойства **TableName**, которое было в компоненте **Table**, в **Query** нет, т.к. таблица, с которой ведется работа, будет указываться в запросах SQL. Поэтому прежде всего надо занести в свойство **SQL** запрос, содержащий имя таблицы, с которой вы хотите работать.

Предупреждение

Прежде, чем начинать детальную настройку компонента **Query**, надо сформировать в его свойстве **SQL** запрос, в котором указывается таблица и перечисляются параметры, если они используются в приложении. Пока такой запрос в **SQL** отсутствует, дальнейшая настройка **Query** невозможна. Запрос, заносимый в **SQL** в начале проектирования, носит чисто служебный характер. В дальнейшем вы можете его программно заменить на любой другой запрос.

Запрос, заносимый вами в **SQL** в начале проектирования, может иметь, например, следующий вид:

```
Select * from pers
```

После этого система поймет, с какой таблицей будет проводиться работа, и можно будет настроить поля в **Query**. Если работа будет проводиться с несколькими таблицами, вы можете все их указать в запросе. Например:

```
Select * from pers, dep
```

После того, как соответствующий запрос написан, можете установить свойство **Active** компонента **Query** в **true**. Если все выполнено правильно, то вы увидите в компоненте **DBGrid1** информацию из запрошенных таблиц.

Можете запустить свое приложение на выполнение и посмотреть его в работе. Оно предоставляет вам возможность просматривать записи, но, к сожалению, не позволяет их редактировать. Это связано с тем, что запрос **Select** возвращает таблицу только для чтения. Впрочем, в таком простом приложении, как наше, это легко исправить. Достаточно установить в компоненте **Query1** свойство **RequestLive** в **true**. Это позволяет возвращать как результат запроса изменяемый, «живой» набор данных, вместо таблицы только для чтения. Точнее, установка **RequestLive** в **true** делает попытку вернуть «живой» набор данных. Успешной эта попытка будет только при соблюдении ряда условий, в частности:

- набор данных формируется обращением только к одной таблице
- набор данных не упорядочен (в запросе не используется **ORDER BY**)
- в наборе данных не используются совокупные характеристики типа **Sum**, **Count** и др.
- набор данных не кэшируется (свойство **CachedUpdates** равно **false**)

В нашем примере все эти условия соблюдаются. Так что можете установить **RequestLive** в **true** и пользователь сможет редактировать данные, удалять записи, вставлять новые записи.

Ваше приложение можно усовершенствовать так же, как вы делали это в главе 9 с приложением на основе **Table**. Для управления отображением данных, как и в компоненте **Table**, имеется уже известный вам Редактор Полей (Field Editor). Вызвать его можно или двойным щелчком на **Query**, или щелчком правой кнопки мыши на **Query** и выбором Fields Editor из всплывающего меню. С Редактором Полей вы уже знакомы (см. раздел 9.5.2 главы 9). В нем вы можете добавить имена получаемых полей (щелчок правой кнопкой мыши и выбор раздела меню Add), задать заголовки полей, отличающиеся от их имен, сделать какие-то поля невидимыми (**Visible**), не редактируемыми (**ReadOnly**), в логических полях можете задать высвечиваемые слова (да;нет), задать формат высвечивания чисел, создать вычисляемые поля, поля просмотра, задать диапазоны значений и многое другое. Если вы создали для полей таблицы словарь (см. раздел 9.6 главы 9), то всего этого вам делать не придется, так как, добавив нужные поля в Редакторе Полей, вы уже будете иметь для них заголовки и все прочие атрибуты.

10.1.6.2 Динамические запросы и параметры Query

Все запросы SQL, которые мы до сих пор рассматривали — это так называемые *статические* запросы. В них фиксировано все: имена таблиц, поля, константы в выражениях и т.п. Но помимо таких статических запросов SQL допускает и *динамические* запросы, использующие параметры. Причем параметры можно применять вместо имен таблиц, имен полей и их значений. Значения этих параметров передаются извне и тем самым, не изменяя текст самого запроса, можно менять возвращаемый им результат.

Параметры задаются в запросе с двоеточием, предшествующим имени параметра:

```
<имя параметра>
```

Например, вы можете записать в запросе **Select** элемент **WHERE** в виде:

```
WHERE Year_b <= :PYear
```

В этом случае вы сравниваете год рождения не с какой-то константой, а со значением параметра, который вы назвали **PYear**.

Теперь обратимся к компоненту **Query**. Если вы введете в его свойство **SQL** запрос, содержащий параметры, например:

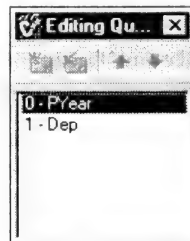
```
Select * from pers where (year_b>:PYear) and (dep=:Dep)
```

а потом щелкнете в Инспекторе Объектов на свойстве **Params**, вам откроется диалоговое окно со списком объектов — указанных вами в запросе параметров **PYear** и **Dep** (см. рис. 10.1). В этом списке вы можете выделять по очереди параметры и в Инспекторе Объектов устанавливать их свойства. Это свойства:

DataType	тип данных параметра (int , string и т.п.)
Name	имя параметра
ParamType	тип параметра (используется при обращении к процедурам, хранимым на сервере — см. раздел 10.3.5.2)
Value	значение параметра по умолчанию
Type — подсвойство Value	тип значения по умолчанию

Рис. 10.1.

Окно задания атрибутов параметров



После того, как вы установили свойства всех параметров, можете использовать их при программировании приложения.

Программный доступ к параметрам во время выполнения приложения осуществляется аналогично доступу к полям набора данных. Свойство **Params** является указателем на массив параметров типа **TParam**, к элементам которого можно обращаться по индексу через его свойство **Items[Word Index]**. Последовательность, в которой располагаются параметры в массиве, определяется последовательностью их упоминания в запросе SQL.

Значения параметров, как и значения полей, определяются такими свойствами объектов — параметров, как **Value**, **AsString**, **AsInteger** и т.п. (см. подробнее в разделе 9.11.4 главы 9 о свойствах похожих объектов — полей). Например, оператор

```
for (int I = 0; I < Query1->Params->Count; I++)
    if (Query1->Params->Items[I]->IsNull &&
        Query1->Params->Items[I]->DataType == ftInteger)
        Query1->Params->Items[I]->AsInteger = -1;
```

задаст значение «-1» всем целым параметрам, которым до этого не было присвоено значение. В нем использовано свойство **Count** — число параметров, свойство **IsNull**, равное **true**, если параметру не задано никакое значение, свойство **DataType**, указывающее тип параметра, и свойство **AsInteger**, дающее доступ к значению параметра как к целому числу.

Другой пример: операторы

```
Query1->Params->Items[0]->AsInteger = 1950;
Query1->Params->Items[1]->AsString = "Бухгалтерия";
```

задают значения первому (индекс 0) и второму (индекс 1) параметрам компонента **Query1**, в свойстве **SQL** которого записан приведенный ранее оператор **Select** с параметрами **:PYear** и **:Dep**. Поскольку в этом операторе параметр **:PYear** упоминается первым, то его индекс равен 0.

Кроме свойства **Items** у **Params** есть еще одно свойство — **ParamValues**. Оно представляет собой массив значений параметров типа **Variant**. В качестве индекса в это свойство передается имя параметра или несколько имен, разделяемых точками с запятой. Например, операторы

```
Query1->Params->ParamValues["PYear"] = 1950;
Query1->Params->ParamValues["Dep"] = "Бухгалтерия";
```

задают те же значения параметрам, что и приведенные выше. Преимуществом является то, что при записи этих операторов не надо помнить индексы параметров. Другим преимуществом свойства **ParamValues** является возможность задать значения сразу нескольким параметрам. Например:

```
Variant par[] = {1950, "Бухгалтерия"};
Query1->Params->ParamValues["PYear;Dep"] = VarArrayOf(par, 1);
```

Другой способ обращения к параметрам, при котором не надо помнить их индексы — использование метода **ParamByName** компонента **Query**. Например, операторы

```
Query1->ParamByName("PYear")->AsInteger = 1950;  
Query1->ParamByName("Dep")->AsString = "Бухгалтерия";
```

задают параметрам с именами **PYear** и **Dep** те же значения, что и приведенные ранее операторы.

Следует оговориться, что задание нового значения параметру само по себе еще не обеспечивает влияния на возвращаемый из запроса результат. Надо повторно выполнить данный запрос, чтобы ощутить изменения. Но о том, как это делается, будет рассказано позднее.

10.1.6.3 Основные свойства Query, связывание таблиц

Большинство свойств **Query** аналогичны свойствам **Table**, рассмотренным ранее (см. раздел 9.5 главы 9). Объекты полей создаются автоматически для тех полей, которые перечислены в операторе SQL. Программный доступ к этим полям осуществляется так же, как в **Table**, с помощью свойства **Fields** (например, **Query1->Fields[0]**) или методом **FieldByName** (например, **Query1->FieldByName("Dep")**). Можно также создавать объекты полей с помощью Редактора Полей, вызываемого двойным щелчком на **Query** или из меню, всплывающего при щелчке на **Query** правой кнопкой мыши. В этом случае доступ к объекту поля можно осуществлять также и по его имени (например, **Query1Dep**).

Предупреждение

При использовании для **Query** Редактора Полей надо добавлять в нем все поля, перечисленные в операторе SQL. Иначе поля, не добавленные в Редакторе Полей, не будут доступны.

Предупреждение

Доступ к полям по имени возможен только в случае, если объекты полей были созданы с помощью Редактора Полей.

Для доступа к значениям полей используются те же их свойства **Value**, **AsString**, **AsInteger** и т.п., что и в **Table**. Точно так же, как в **Table**, можно осуществлять навигацию по набору данных, устанавливать фильтры, ограничивать вводимые значения полей, кэшировать изменения.

Из свойств, отличных от **Table**, остановимся на свойстве **DataSource**. Это свойство позволяет строить приложения, содержащие связанные друг с другом таблицы. Рассмотрим, как это делается, на том же примере, который использовался в главе 9 при рассмотрении **Table**. Пусть мы хотим построить приложение, включающее в себя таблицу **Dep**, содержащую список отделов (в поле **Dep**) и их характеристику, в качестве головной таблицы и таблицу персонала **Pers**, содержащую в поле **Dep** имя отдела, в котором работает каждый сотрудник. Мы хотим, чтобы при выборе записи в таблице **Dep** в таблице **Pers** отбирались только записи, относящиеся к выбранному отделу.

Откройте новое приложение. Перенесите на форму компоненты **Query1**, **DataSource1**, **DBGrid1** и соедините их обычной цепочкой: в **DBGrid1** задайте свойство **DataSource** равным **DataSource1**, а в **DataSource1** задайте свойство **DataSet** равным **Query1**. Компонент **Query1** настройте на таблицу **Dep**. Для этого установите свойство **DatabaseName** (например, **dbP**), а в свойстве **SQL** напишите оператор

```
Select * from Dep
```

Установите свойство **Active** в **true** и убедитесь, что все работает нормально: в **DBGrid1** должно отобразиться содержимое таблицы **Dep**.

Создайте другую аналогичную цепочку, перенеся на форму компоненты **Query2**, **DataSource2**, **DBGrid2**, и свяжите ее с таблицей **Pers** запросом

```
Select * from Pers
```

в компоненте **Query2**. Установите свойство **Active** компонента **Query2** в **true** и в **DBGrid2** должно отобразиться содержимое таблицы **Pers**.

Вы можете запустить приложение и убедиться, что оно работает, но таблицы независимы. Теперь давайте свяжем эти таблицы. Делается это следующим образом. Измените текст запроса в свойстве **SQL** вспомогательного компонента набора данных **Query2** на

```
Select * from Pers where (Dep=:Dep)
```

В этом запросе вы указываете условие отбора: значение поля **Dep** должно быть равно параметру **:Dep**. В предыдущем разделе было рассказано, как вводить в запрос и использовать параметры. Но в данном случае не надо определять этот параметр с помощью редактора параметров, вызываемого из свойства **Params** компонента **Query2**. Вместо этого в свойстве **DataSource** компонента **Query2** надо сослаться на **DataSource1** — источник данных, связанный с таблицей **Dep**. Это скажет приложению, что оно должно взять значения параметра **:Dep** из текущей записи этого источника данных. А поскольку имя параметра совпадает с именем поля в источнике данных, то в качестве значения параметра будет взято текущее значение этого поля. Таким образом, вспомогательная таблица, запрашиваемая в **Query2**, оказывается связанной с головной таблицей, запрашиваемой в **Query1**.

После изменения содержимого свойства **SQL** свойство **Active** компонента **Query2** сбросится в **false**. Установите его в **true** и запустите приложение. Вы увидите, что при перемещении по первой таблице во второй отображаются только те записи, которые относятся к отделу, указанному в текущей записи первой таблицы.

10.1.6.4 Основные методы компонента Query

К основным методам **Query** можно отнести методы открытия и закрытия соединения с базой данных.

Метод **Close** закрывает соединение с базой данных, переводя свойство **Active** в **false**. Этот метод надо выполнять перед изменением каких-то свойств, влияющих на выполнение запроса или на отображение данных. Например, при изменении параметров запроса в свойстве **SQL** надо сначала методом **Close** закрыть соединение, связанное с прежним запросом, а потом уже выполнять новый запрос.

Метод **Open** открывает соединение с базой данных и выполняет запрос, содержащийся в свойстве **SQL**. Но этот метод применим только в том случае, если запрос сводится к оператору **Select**. Если же запрос содержит другой оператор, например, **Update** или **Insert**, то при выполнении **Open** будет генерироваться исключение **EDatabaseError**. Для осуществления любого другого запроса, кроме **Select**, используется метод **ExecSQL**. Он подготавливает выполнение данного запроса, если подготовка не осуществлена заранее, и затем выполняет его. Подготовка выполнения требует определенных затрат времени. Поэтому для ускорения взаимодействия с базой данных полезно перед первым выполнением запроса выполнить метод **Prepare**.

При изменении текста в свойстве **SQL** методы **Close** и **Prepare** вызываются автоматически.

Наличие двух методов выполнения запроса — **Open** и **ExecSQL** ставит вопрос: «Как же поступить, если запрос сформирован пользователем или программой и его характер неизвестен?». Неизвестный запрос, сформированный в некоторой переменной **ssql** типа строки, можно выполнить с помощью следующего кода:

```
try
{
  Query1->SQL->Clear();
  Query1->SQL->Add(ssql);
```



```
Query1->Open();  
}  
catch (EDatabaseError&)  
{  
    Query1->ExecSQL();  
}
```

В блоке **try** (см. раздел 12.10.5 главы 12) осуществляется попытка выполнить запрос с помощью метода **Open**. А если эта попытка завершается генерацией исключения, т.е. если запрос состоит из оператора, отличного от **Select**, то генерируется исключение **EDatabaseError**, которое перехватывается в разделе **catch**, и выполняется метод **ExecSQL**.

В некоторых случаях приложению требуется получить список имен полей таблицы, связанной с **Query**. Это может быть сделано методом **GetFieldNames**, который загружает список имен полей в любую переменную типа **TStrings**, передаваемую в него в качестве аргумента. Например, оператор

```
Query1->GetFieldNames(ComboBox1->Items);
```

загружает в выпадающий список **ComboBox1** имена полей таблицы, связанной с **Query1**. В дальнейшем будет приведен пример использования этого метода.

10.16.5 Кэширование изменений, совместное применение Query и UpdateSQL

Метод **ExecSQL** осуществляет немедленную модификацию таблицы. Однако нередко удобнее было бы кэшировать изменения (хранить их временно в памяти), а после того, как все изменения и проверки сделаны, переслать их в базу данных или, по решению пользователя, отменить все сделанные исправления.

Это делается так же, как и для компонента **Table** (см. раздел 9.11.3 главы 9): устанавливается в **true** свойство **CachedUpdates** компонента **Query** и применяются методы **ApplyUpdates** для записи изменений в базу данных, метод **CancelUpdates** для отмены изменений и метод **CommitUpdates** для очистки буфера кэша.

Но режим кэширования позволяет сделать большее — он позволяет подключить в приложение компонент **UpdateSQL**. Этот компонент, расположенный на странице библиотеки **Data Access**, позволяет модифицировать наборы данных, открытые в режиме только для чтения. Это особенно важно для наборов данных, открываемых **Query** с запросом **Select**, поскольку **Select** создает таблицу только для чтения.

Давайте построим приложение, демонстрирующее режим кэширования и тождественное тому, которое рассматривалось в разделе 9.11.3 главы 9. Возьмите то же самое приложение (рис. 9.47), только замените в нем компонент **Table1** на компонент **Query1** и везде в тексте тоже проведите соответствующую замену **Table1** на **Query1**. В свойстве **SQL** компонента **Query1** запишите «**Select * from Pers**» и установите свойство **CachedUpdates** в **true**.

Запустите свое приложение, и вы увидите, что оно, увы, не работает. Отредактировать запись или вставить новую запись невозможно. А из кнопок навигатора доступны только кнопки навигации. Причина всего этого была указана ранее — **Query** с запросом **Select** создает таблицу только для чтения.

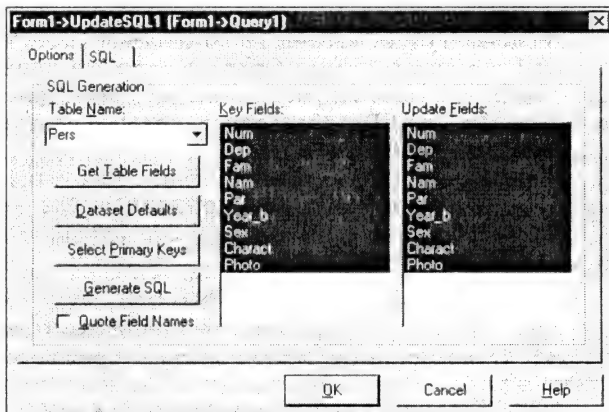
Давайте исправим ваше приложение. Поместите на него компонент **UpdateSQL** со страницы библиотеки **Data Access**. Чтобы связать его с приложением, установите в компоненте **Query1** свойство **UpdateObject** равным имени введенного компонента **UpdateSQL1**. Это имя вы можете выбрать из выпадающего списка в свойстве **UpdateObject**.

Теперь можно задавать свойства компонента **UpdateSQL1**. Этих свойств, кроме **Name** и **Tag**, всего 3: **DeleteSQL**, **InsertSQL** и **ModifySQL**. Они содержат соответственно запросы, которые должны выполняться при удалении, вставке или модификации записи. Эти запросы можно записать обычным образом, вручную. А можно воспользоваться редактором **UpdateSQL**, который вызывается двойным

щелчком на **UpdateSQL** или при выделенном **UpdateSQL** вызывается из всплывающего меню. Окно этого редактора имеет вид, представленный на рис. 10.4. Первая страница Options редактора UpdateSQL, представленная на рисунке, содержит два окна Key Fields и Update Fields.

Рис. 10.2.

Окно редактора UpdateSQL



В левом окне Key Fields надо выделить поля, по которым программа будет распознавать модифицируемую или удаляемую запись. Можно выделить все поля, что гарантирует максимально возможную надежность распознавания, но можно ограничиться выбором нескольких наиболее важных полей. При работах с очень большими таблицами это сэкономит время выполнения запроса. Для выбора совокупности полей можно нажать клавишу Ctrl и, не отпуская ее, выбрать курсором мыши нужные поля.

В правом окне Update Fields надо выделить поля, значения которых будут задаваться при модификации или вставке записи. Обычно целесообразно выделить в этом окне все поля, хотя, конечно, могут быть случаи, когда не все поля надо задавать. В частности, нельзя задавать вычисляемые поля, если они имеются в таблице.

После того, как вы выделили в этих окнах поля, нажмите кнопку Generate SQL. После этого вам будет показана вторая страница SQL редактора UpdateSQL, на которой вы можете просмотреть сгенерированные запросы для модификации (Modify), вставки (Insert) и удаления (Delete) записи. После щелчка на OK эти запросы перенесутся в свойство **DeleteSQL**, **InsertSQL** и **ModifySQL**, где вы их можете дополнительно отредактировать.

При всех выделенных полях запрос **ModifySQL** будет иметь вид:

```
update Pers
set
  Num = :Num,
  Dep = :Dep,
  Fam = :Fam,
  Nam = :Nam,
  Par = :Par,
  Year_b = :Year_b,
  Sex = :Sex,
  Charact = :Charact,
  Photo = :Photo
where
  Num = :OLD_Num and
  Dep = :OLD_Dep and
  Fam = :OLD_Fam and
  Nam = :OLD_Nam and
  Par = :OLD_Par and
```

```
Year_b = :OLD_Year_b and  
Sex = :OLD_Sex and  
Charact = :OLD_Charact and  
Photo = :OLD_Photo
```

В нем в разделе **Set** указана установка всех полей в значения, задаваемые соответствующими параметрами с именами, тождественными именам полей. В этот раздел включаются те поля, которые вы выделили в окне Update Fields редактора UpdateSQL. Заполнение этих параметров при выполнении соответствующих команд приложения вам не потребуется: все это сделают автоматически методы компонента UpdateSQL. В разделе **Where** содержатся условия, по которым идентифицируется модифицируемая запись. В этих условиях используются параметры с именами, тождественными именам полей, но с префиксом **OLD_**. Эти параметры — прежние значения соответствующих полей, которые были получены компонентом до модификации записи. В условия **Where** включены те поля, которые вы выделили в окне Key Fields редактора UpdateSQL.

Естественно, что в зависимости от приложения вы можете заменить эти автоматически сгенерированные имена параметров на другие или вообще использовать запросы без параметров. Только имейте в виду, что значения автоматически сгенерированных параметров в дальнейшем будут автоматически загружаться из объектов-полей, а если вы от них откажетесь, то и соответствующие значения должны будете задавать сами.

Предупреждение

Если вы измените имена параметров в запросах SQL компонента UpdateSQL, вам придется программно задавать значения этих параметров. Если же вы оставите автоматически сгенерированные имена, то в дальнейшем значения параметров будут автоматически загружаться из объектов-полей. Так что изменять имена параметров неразумно.

Для того, чтобы запросы компонента UpdateSQL срабатывали, все объекты полей, имена которых в них используются, должны быть или автоматически сгенерированы (т.е. без применения в компоненте Query Редактора Полей), или введены в Редакторе Полей. В противном случае при попытке выполнить запрос вам будет выдано сообщение: «Field ... is of an unknown type» (поле ... неизвестного типа) и запрос не выполнится.

Предупреждение

В запросах компонента UpdateSQL должны фигурировать только те поля, которые введены вами в Редакторе Полей компонента Query, или вы не должны пользоваться в Query Редактором Полей.

Посмотрим на приведенный выше сгенерированный запрос **ModifySQL** с точки зрения нашего приложения. Прежде всего очевидно, что из условия запроса **where** надо удалить поля **Charact** и **Photo**, так как сравнение полей такого типа бессмысленно. Вообще для нашей таблицы достаточно оставить сравнение только по полю **Num**, поскольку это поле обеспечивает уникальность каждой записи. Кроме того из раздела **set** надо исключить поле **Num**, так как оно типа **Autoincrement**, а значения полей этого типа устанавливать нельзя: они автоматически нарастают с каждой новой записью. Имеет также смысл исключить поля **Charact** и **Photo**, так как они в нашем приложении не отображаются. Таким образом, запрос **ModifySQL** имеет смысл оставить в виде:

```
update Pers  
set  
  Dep = :Dep,  
  Fam = :Fam,
```

```
Nam = :Nam,
Par = :Par,
Year_b = :Year_b,
Sex = :Sex
where
  Num = :OLD_Num
```

Запрос в свойстве **DeleteSQL** строится по такому же принципу. Его можно записать в виде

```
delete from Pers
where
  Num = :OLD_Num
```

Запрос **InsertSQL**, построенный при выделении всех полей в окне рис. 10.2, имеет вид:

```
insert into Pers
(Num, Dep, Fam, Nam, Par, Year_b, Sex, Charact, Photo)
values
(:Num, :Dep, :Fam, :Nam, :Par, :Year_b, :Sex,
:Charact, :Photo)
```

Его, очевидно, тоже надо изменить, убрав задание поля **Num**, поскольку оно увеличивается автоматически и не может изменяться приложением. Имеет смысл убрать также поля **Charact** и **Photo**, так как они не фигурируют в нашем приложении. В итоге запрос приобретет следующий вид:

```
insert into Pers
(DEP, FAM, NAM, PAR, YEAR_B, SEX)
values
(:DEP, :FAM, :NAM, :PAR, :YEAR_B, :SEX)
```

Осталось несколько изменить по сравнению с приложением на основе **Table** обработчик события **OnClick** кнопки Фиксация. Он должен иметь вид:

```
Query1->ApplyUpdates();
Query1->CommitUpdates();
Query1->Close();
Query1->Open();
modif = false;
```

По сравнению с тем, что было раньше, в него добавлены операторы закрывания (**Close**) и открывания (**Open**) соединения с базой данных компонента **Query1**. Это желательно сделать, чтобы в **Query1** отобразилось новое состояние базы данных после внесенных в нее изменений. Если не предусмотреть этого, то, например, будет невозможно вставить в сеансе работы новую запись, подтвердить изменение, а затем удалить эту запись. Дело в том, что если мы не обновили данные в **Query1**, то в этих данных отсутствует вставленная в данном сеансе работы запись. И, следовательно, ее не удастся удалить (можете проверить это в эксперименте).

Запустите теперь приложение, и вы увидите, что оно стало работать так же, как работало ранее с компонентом **Table**. Можно редактировать записи, удалять, вставлять новые записи, управлять кэшированием.

В данном случае все получилось так просто, поскольку соответствующие методы работы с **UpdateSQL** автоматически выполняются компонентами **DBNavigator** и **DBGrid**. В некоторых других приложениях эти методы надо вызывать явно. Поэтому коротко ознакомимся с ними.

Метод

```
Apply(Db::TUpdateKind UpdateKind)
```

обеспечивает загрузку значений всех необходимых параметров и выполнение одного из запросов компонента **UpdateSQL**. Какого именно — определяется параметром **UpdateKind**, который может принимать значения **ukModify**, **ukInsert** или

ukDelete, что соответствует выполнению запроса, хранящегося в свойствах **ModifySQL**, **InsertSQL** и **DeleteSQL**.

Например, оператор

```
UpdateSQL1->Apply(ukModify)
```

вызовет выполнение запроса, содержащегося в свойстве **ModifySQL**.

Если нужный запрос **SQL** не содержит параметров, то эффективнее вызвать такой метод компонента **UpdateSQL**, как **ExecSQL**:

```
ExecSQL(Db::TUpdateKind UpdateKind)
```

который тоже обеспечивает выполнение указанного запроса, но без предварительной загрузки значений параметров. Наконец, метод

```
SetParams(Db::TUpdateKind UpdateKind)
```

обеспечивает загрузку значений параметров указанного запроса **SQL** без его выполнения. Таким образом, метод **Apply** — это просто последовательное выполнение методов **SetParams** и **ExecSQL**.

10.1.7 Пример формирования произвольных запросов SQL

Вы можете попробовать использовать компонент **Query** вместо **Table** в любом из приложений, рассмотренных в главе 9. Большое количество подобных примеров приложений с **Query** вы можете найти в книге [8]. Несколько примеров имеется на прилагаемом к данной книге диске. Целесообразность замены **Query** на **Table**, как было сказано в разделе 10.1.6.1, зависит от того, работаете ли вы с локальными базами данных, или с удаленным сервером. Но есть приложения, в которых пользователю разрешается формировать любые запросы к базе данных. В таких приложениях без языка **SQL** и, соответственно, без компонентов **Query** не обойтись. Рассмотрим на чисто демонстрационном примере, как строить подобные приложения.

Общий вид приложения, которое мы с вами попробуем построить, приведен на рис. 10.3. Вы можете найти его на приложенном к книге диске.

Рис. 10.3.
Пример приложения, позволяющего
пользователю формировать
произвольные запросы к базе данных



Приложение позволяет пользователю сформировать различные запросы **Select** к таблице **Pers**. Выпадающий список **Поле** (типа **TComboBox**, в программе назван **CBFields**) заполнен именами полей, которые пользователь может выбирать из него при формировании запроса. Выпадающий список **Операции**, знаки (типа **TComboBox**, в программе назван **CBOp**) заполнен символами допустимых операций, функций и знаков, которые пользователь также может выбирать при формировании запроса. Окно, расположенное ниже кнопок, является компонентом типа **TMemo** (в программе названо **MSQL**). Это окно служит для отображения формируемого запроса **SQL**. Ниже расположена таблица **DBGrid1** типа **TDBGrid**, которая через компонент **DataSource** связана с компонентом **Query** типа **TQuery**. Этот компонент и выполняет сформированные пользователем запросы.

Кнопка **Новый запрос** начинает формирование запроса, посылая в **MSQL** текст «**Select** ». Затем пользователь может вводить имена полей или непосредственно в текст, или, чтобы не ошибиться, выбирая их имена из списка **Поле**. При вводе совокупных характеристик или вычисляемых полей пользователь может брать необходимые символы и ключевые слова из списка **Операции**, знаки. При желании пользователь может щелкнуть на кнопке **Условие** (в запрос вводится элемент **WHERE**), на кнопке **Порядок** (в запрос вводится элемент **ORDER BY**) или кнопке **Группировка** (в запрос вводится элемент **GROUP BY**) и сформировать условия отбора, сортировки, группирования. После того, как запрос сформирован, пользователь выполняет его щелчком на кнопке **Выполнить** и в окне отображаются результаты запроса.

Ниже приведен полный текст данного приложения.

```
// Перечислимый тип, определяющий режим работы
// в каждый момент времени

enum TRegim {RNone, RFields, RWhere, ROrder, REnd} Regim;

//-----
void __fastcall TForm1::ADDS(String s)
{
    // Добавление в конец последней строки в Мето новой строки s
    MSQL->Lines->Strings[MSQL->Lines->Count-1] =
        MSQL->Lines->Strings[MSQL->Lines->Count-1] + s;
}

//-----
void __fastcall TForm1::CBFieldsChange(TObject *Sender)
{
    if ((Regim == REnd) || (MSQL->Lines->Count < 1))
        ShowMessage(
            "Начните новый запрос или вводите оператор вручную");
    else ADDS(" "+CBFields->Text);
}

//-----
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    // Загрузка в выпадающий список имен полей таблицы
    Query->GetFieldNames(CBFields->Items);
    CBFields->Items->Insert(0, "");
    CBFields->ItemIndex = 0;
    CBOp->ItemIndex = 0;
    Regim = RNone;
}

//-----
void __fastcall TForm1::BbeginClick(TObject *Sender)
{
    MSQL->Clear();
    MSQL->Lines->Add("Select ");
}
```

```

    Regim = RFields;
}
//-----
void __fastcall TForm1::BexecClick(TObject *Sender)
{
    if (Regim == RNone)
    {
        ShowMessage("Вы не ввели запрос");
        return;
    }
    if (Regim == RFields)
        ADDS(" FROM PERS");
    Regim = REnd;
    Query->SQL->Assign(MSQL->Lines);
    Query->Open();
}
//-----
void __fastcall TForm1::BWhereClick(TObject *Sender)
{
    if (Regim == RFields)
        ADDS(" FROM PERS");
    ADDS(" WHERE");
    Regim = RWhere;
}
//-----
void __fastcall TForm1::CBOpChange(TObject *Sender)
{
    if ((Regim == REnd) || (MSQL->Lines->Count < 1))
        ShowMessage("Начните новый запрос или вводите оператор вручную");
    else ADDS(" "+CBOp->Text);
}
//-----
void __fastcall TForm1::BOrderClick(TObject *Sender)
{
    if (Regim == RFields)
        ADDS(" FROM PERS");
    ADDS(" ORDER BY");
    Regim = ROrder;
}
//-----
void __fastcall TForm1::BGroupClick(TObject *Sender)
{
    if (Regim == RFields)
        ADDS(" FROM PERS");
    ADDS(" GROUP BY");
    Regim = ROrder;
}
}

```

Сделаем некоторые комментарии к этому тексту. В начале вводится переменная **Regim** перечислимого типа:

```
enum TRegim {RNone, RFields, RWhere, ROrder, REnd} Regim;
```

возможные значения которой определяют, в каком режиме в данный момент находится приложение. В начале значение переменной **Regim** задается равным **RNone**. Обработчики щелчков всех кнопок проверяют значение переменной **Regim** и в зависимости от результатов этой проверки производят те или иные операции. Кроме того они изменяют значение этой переменной, сообщая приложению, какие операции выполняет пользователь. Например, обработчик щелчка кнопки Выполнить в процедуре **BexecClick** проверяет **Regim** и, если значение этой переменной оказывается **RNone**, это означает, что пользователь, не нажав до этого ни одной кнопки, сразу щелкнул на кнопке Выполнить. В этом случае пользователю выдается замеча-

ние «Вы не ввели запрос» и обработка события прерывается. Соответствующие проверки режима вы можете увидеть и в других обработчиках событий.

Далее в приложении вводится процедура **ADDS**, которая добавляет заданную текстовую строку в конец текста, содержащегося в **MSQL**. Это сделано просто во избежание повтора входящего в эту процедуру оператора во многих местах кода. Конечно, при этом в заголовочном файле модуля добавляется в описание класса соответствующее объявление этой процедуры:

```
void __fastcall ADDS(String s);
```

Процедура **FormCreate** является обработчиком события **OnCreate** формы. В этой процедуре производится загрузка списка **CBFields** значениями имен полей в таблице. Это очень легко делается методом **GetFieldNames**, который загружает список имен полей в любую переменную типа **TStrings**, передаваемую в него в качестве аргумента.

Остальные процедуры приложения, вероятно, специальных комментариев не требуют. Просмотрите их, проверьте, как работает приложение, и все неясности разрешатся.

10.2 Работа с базами данных в сети

10.2.1 Транзакции и проблемы многопользовательского режима работы

Все, что рассматривалось ранее, и все созданные вами приложения могут работать и с локальными базами данных, и в сети с базами с разделенными файлами (с одновременным доступом нескольких пользователей), и с удаленным сервером на платформе клиент/сервер. Однако, при работе в сети, когда к одной и той же базе данных одновременно может обращаться несколько пользователей, может возникнуть множество проблем. Например, вы прочитали какую-то запись из таблицы и редактируете ее. Но, пока вы редактируете, другой пользователь может изменить эту запись или вообще удалить ее. Что тогда будет при попытке сохранить ваши изменения в этой уже отредактированной или вообще удаленной записи?

Эта и множество других аналогичных проблем разрешаются с помощью механизма транзакций (transaction). *Транзакция* — это групповая операция, связанная с передачей сообщения. С точки зрения приложения транзакция — это группа логически связанных операторов **SQL**, причем только успешное выполнение всех операторов должно приводить к изменению данных сервером. Пока все операторы транзакции не выполнены, сохраняется возможность отменить их и не фиксировать результаты в базе данных. Подобный механизм необходим для надежной работы с удаленным сервером и в многопользовательском режиме. В тех приложениях, которые вы создавали до сих пор, транзакции тоже использовались, но неявно. Каждый вызов метода **Post**, каждое изменение таблицы автоматически начинало и заканчивало транзакцию. Но часто такой автоматизм недостаточен и не эффективен. Обычно надо управлять транзакциями явно. Как это делается, мы рассмотрим ниже. Но сначала попробуем осмыслить проблемы, возникающие при совместном доступе пользователей к одной и той же таблице. Их можно систематизировать следующим образом:

- *Чтение незафиксированных изменений* происходит, если одна транзакция прочла изменения, сделанные другой транзакцией, но еще не зафиксированные в базе данных. Если эти изменения будут отменены вносившей их транзакцией, то окажется, что прочитавшая их транзакция будет работать с ошибочными данными.

- *Неповторяемое чтение данных* происходит, если в момент чтения данных одной транзакцией другая транзакция изменяет эти данные. В этом случае повторное чтение тех же данных невозможно.
- *Фантомные записи* возникают, если одна транзакция прочла незафиксированные новые записи, созданные другой транзакцией (может быть эти записи и не будут вставлены в базу данных), или прочла записи, которые к моменту ее завершения уже удалены из таблицы другой транзакцией. В обоих случаях первая транзакция оперирует с фантомами, которых в таблице нет.
- *Потерянные изменения* возникают, если одна транзакция модифицирует изменения, сделанные другой транзакцией.
- *Вторичные эффекты модификации* могут происходить в базах данных, в которых значения одних записей зависят от значений других записей. Тогда наложения результатов одновременной работы нескольких транзакций возможны даже в случае, если транзакции оперируют с разными записями.

10.2.2 Управление транзакциями, компонент Database

Выше упоминалось о том, что даже без специального, целенаправленного управления транзакциями они в действительности автоматически осуществляются при каждой модификации данных. Это производится компонентом типа **TDatabase**, который C++Builder включает автоматически в любое приложение, работающее с базами данных. Этот компонент решает следующие задачи:

- Создание соединения с удаленным сервером
- Регистрация пользователя при первом обращении к серверу
- Создание локальных псевдонимов приложений
- Управление транзакциями
- Определение уровня изоляции транзакции (регулирование одновременных транзакций к одним и тем же таблицам)

Если же вы хотите сознательно управлять транзакциями, вы должны явным образом включить компонент **Database** в свое приложение. Он расположен в библиотеке на странице Data Access.

Database связывается с компонентами наборов данных **Table**, **Query** и другими через имя базы данных, к которой он подключается. Это имя задается в свойстве **DatabaseName**. Может быть задан псевдоним базы данных или полный путь к ней. Если задается база данных, имеющая псевдоним BDE, то свойства **AliasName**, **DriverName** и **Params** можно не задавать. В противном случае надо задать или свойство **AliasName**, или свойства **DriverName** и **Params**.

Установку значений всех этих свойств можно проводить непосредственно в Инспекторе Объектов, но удобнее воспользоваться специальным редактором, который вызывается двойным щелчком на **Database** (рис. 10.4). Как уже говорилось, вы можете ограничиться заданием только имени базы данных (окно Name). Но если ваша база данных имеет псевдоним, вы можете указать и его (окно Alias Name). В этом случае можно щелкнуть на кнопке Defaults (умолчание) и в окне Parameter overrides появятся значения параметров по умолчанию. Вы можете внести в них какие-то изменения. Например, при работе с базой данных, защищенной паролем, вы можете указать в параметрах этот пароль (на рис. 10.4 указан пароль «1») и сбросить индикатор login prompt — приглашение к соединению, которое запрашивает пароль. Тогда при запуске вашего приложения не будет каждый раз запрашиваться пароль. Это облегчит работу пользователей, но, конечно, снимет защиту вашей базы данных.

Индикатор Keep inactive connection устанавливает свойство **KeepConnection**, о котором будет сказано ниже.

Рис. 10.4.

Установка значений свойств компонента Database

После всех установок в редакторе щелкните на **OK** и введенные вами установки заполняют значения свойств **DatabaseName**, **AliasName**, **DriverName**, **Params** и **KeepConnection**.

Свойство **Connected** (соединение) совместно со свойством **KeepConnection** управляют процессом соединения компонентов с базой данных. Если **KeepConnection** равно **true**, то соединение с базой данных постоянное даже при отсутствии открытых наборов данных. Если же **KeepConnection** равно **false**, то для регистрации на сервере надо устанавливать **Connected** в **true** при каждом открытии таблицы.

Свойство **TransIsolation** определяет уровень изоляции транзакции. Это свойство может иметь значения:

tiDirtyRead	Позволяет читать все текущие изменения, проводимые другими транзакциями до их фиксации
tiReadCommit	Позволяет читать только зафиксированные изменения, проводимые другими транзакциями. Это значение принято по умолчанию
tiRepeatableRead	После начала транзакции не позволяет читать даже подтвержденные изменения, проводимые другими транзакциями в прочитанных данных. Следовательно, при повторном прочтении на протяжении данной транзакции той же записи будут получены прежние результаты, даже если другие транзакции их уже изменили

Если обратиться к сформулированным в предыдущем разделе проблемам, возникающим при совместной работе транзакций с одними и теми же данными, то можно заметить, что только значение **tiRepeatableRead** исключает проблемы чтения незафиксированных изменений, неповторяемого чтения данных и появления фантомных записей. Принятый по умолчанию уровень **tiReadCommit** исключает только проблему неповторяемого чтения данных.

Указанные выше значения свойства **TransIsolation** могут не поддерживаться на конкретном сервере, с которым вы работаете. В этом случае сервер переходит на доступный ему более высокий, чем запрошенный, уровень изоляции. Ниже приведена таблица уровней для различных систем.

TransIsolation	InterBase	Oracle	Sybase & Microsoft
tiDirtyRead	Read committed	Read committed committed	Read
tiReadCommit	Read committed	Read committed	Read committed
tiRepeatableRead	Repeatable Read	Repeatable Read (READ ONLY)	Error (не поддерживается)

Начало транзакции осуществляется методом **StartTransaction** компонента **Database**. При этом начинающаяся транзакция использует текущее значение свойства **TransIsolation** для определения уровня изоляции.

Завершается транзакция методом **Commit**, фиксирующим ее результаты в базе данных. Метод **Rollback** можно использовать для «отката» назад при неудаче — этот метод отменяет все операции с базой данных, выполненные после последнего выполнения метода **Commit**.

Таким образом, программа работы с данными должна строиться по следующей схеме:

```
Databasel->StartTransaction();
```

Группа операторов изменения данных (ExecSQL и др.)

```
Проверка результатов: Если успешно — Databasel->Commit();
                      Если неудача — Databasel->Rollback();
```

В случае выполнения **Rollback** все изменения, сделанные на протяжении транзакции отменяются. Таким образом, модификации данных, предусмотренные в транзакции, или будут выполнены все, или не будет выполнено ничего.

Из общих рекомендаций по организации транзакций можно прежде всего отметить рекомендацию по возможности сокращать число транзакций. Сокращение числа транзакций возможно за счет команд SQL групповой обработки записей. Например, при необходимости переименовать подразделение во всех записях таблицы можно написать следующий код (предполагается, что мы хотим переименовать «Цех 1» в «Цех 2»):

```
Table1->First();
while (! Table1->Eof)
{
    if (Table1->FieldByName("Dep")->AsString == "Цех 2")
    {
        Table1->Edit();
        Table1->FieldByName("Dep")->AsString = "Цех 1";
    }
    Table1->Next();
}
Table1->First();
```

Этот код предполагает осуществление столько транзакций, в скольких записях будут сделаны исправления. Для каждого исправления будет генерироваться своя команда **Update**. Если ваша база данных большая, то это может привести к значительным затратам времени и к проблемам на сервере при регистрации транзакций. А если во время выполнения цикла возникли какие-то исключительные ситуации, то часть записей будет изменена, а часть — нет. Последнее можно исправить, поместив перед циклом оператор **StartTransaction**. Но неэффективность выполнения останется.

Более удачным решением в данном случае является выполнение с помощью **Query** команды типа:

```
UPDATE Pers SET Dep='Цех 1' WHERE Dep='Цех 2'
```

Эта команда будет связана всего с одной транзакцией и гарантирует, что или все необходимые изменения будут сделаны, или не будет сделано ни одного, если в процессе выполнения возникнут какие-то проблемы.

Число транзакций сокращается также за счет описанного ранее кэширования изменений, поскольку в этом случае все локально хранящиеся изменения заносятся в базу данных одной транзакцией.

Наряду с сокращением числа транзакций желательно уменьшать длительность каждой, поскольку слишком долго выполняемая транзакция может надолго заблокировать ресурсы для других приложений. Например, если таблица Pers, с которой вы работали в предыдущем примере, очень большая, то приведенную выше команду можно разбить на две, выполняемые в пределах отдельных транзакций:

```
UPDATE Pers SET Dep='Цех 1'
WHERE (Num BETWEEN 1 AND 10000) and (Dep='Цех 2')
```

и

```
UPDATE Pers SET Dep='Цех 1'
WHERE (Num > 10000) and (Dep='Цех 2')
```

10.2.3 Работа с SQL Monitor

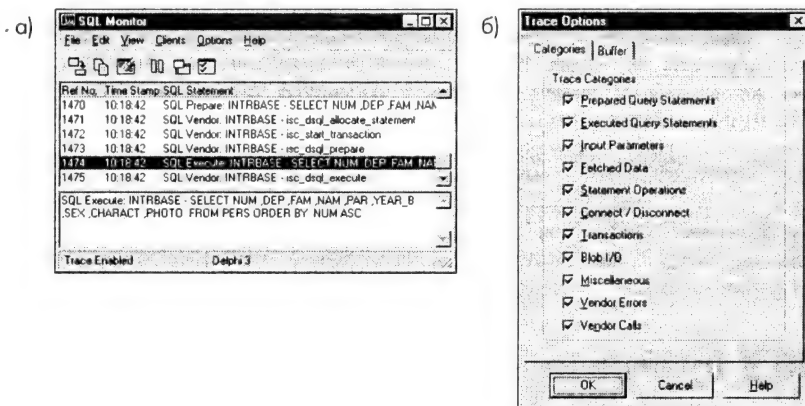
Удобным инструментом для отладки программ, работающих с базами данных, является **SQL Monitor**, запускаемый командой главного меню Database | SQL Monitor. Окно **SQL Monitor** показано на рис. 10.5 а. Осуществляя постоянный мониторинг обмена сообщениями с сервером, **SQL Monitor** позволяет просматривать команды **SQL**, которые неявным для пользователя образом передаются через **SQL Links** удаленным серверам баз данных. Двойной щелчок на том или ином сообщении позволяет развернуть или свернуть его полный текст внизу окна. Командой меню Options | Always on Top можно установить опцию, обеспечивающую постоянное присутствие окна **SQL Monitor** на экране поверх всех других окон.

Раздел меню Options | Trace Options открывает окно опций (рис. 10.5 б), содержащее следующие индикаторы, определяющие отображение тех или иных сообщений:

Prepared Query Statements	Команды, подготовленные к отправке на сервер
Executed Query Statements	Команды, выполняемые на сервере
Statement Operations	Такие операции как ALLOCATE , PREPARE , EXECUTE , FETCH
Connect / Disconnect	Операции при соединении и отключении от базы данных (распределение памяти и т.п.)
Transactions	Действия при обработке транзакций: BEGIN , COMMIT , ROLLBACK , ABORT
BLOB I/O	Действия над BLOB (бинарными) полями
Miscellaneous	Другие операции с базами данных, не перечисленные ранее
Vendor Errors	Сообщения об ошибках, возвращаемые сервером
Vendor Calls	Вызовы функций API сервера

Рис. 10.5.

Окно программы SQL Monitor (а) и окно ее опций (б)



При начальной отладке приложения, вероятно, полезно установить только опции Executed Query Statements и Transactions. Они позволяют увидеть логику работы. В дальнейшем для оптимизации работы с базой данных можно подключить и другие опции.

10.2.4 Управление доступом

Системы управления доступом должны обеспечивать эффективное взаимодействие приложений, работающих одновременно с базой данных. Исключить обсуждавшиеся выше наложения результатов работы различных транзакций можно путем блокировки ресурсов, необходимых выполняемой транзакции. Тогда другие транзакции, обращающиеся к тем же ресурсам, будут ждать, пока данная транзакция не завершится, после чего ресурсы будут освобождены. Правда, при этом возможна взаимная блокировка, когда одна транзакция заблокировала ресурсы, необходимые другой, а эта другая заблокировала какие-то ресурсы, необходимые первой. Это тупиковая ситуация, которую можно разрешить только извне.

C++Builder реализует более оптимальное управление доступом. Оно предполагает, что большинство обращений к базе данных — это обращения для чтения. И очень невелика вероятность того, что два приложения одновременно будут изменять одну и ту же запись и одни и те же поля в ней. Такое редкое наложение изменений будет восприниматься как аварийная ситуация, приводящая к генерации исключений.

Процесс работы приложения C++Builder с базой данных выглядит следующим образом. При выполнении приложения оно получает копию записи с сервера, позволяет что-то в ней изменить, а затем посылает изменения на сервер с помощью команды **Update**. Эта команда SQL имеет элемент **Where**, перечисляющий значения полей, совпадение которых идентифицирует изменяемую запись. Эти значения берутся из копии записи и, следовательно, не зависят от того, не изменились ли за это время поля реальной записи.

В компонентах **Table** и **Query** имеется свойство **UpdateMode** — режим обновления. Значения этого свойства и определяют, какие поля будут включены в элемент **Where** команды **Update**. Возможные значения:

- upWhereAll** **Where** включает все поля записи (принято по умолчанию)
- upWhereChanged** **Where** включает ключевые поля и поля, которые были изменены
- upWhereKeyOnly** **Where** включает только ключевые поля

Вариант **upWhereAll** наиболее надежен, поскольку распознает запись по значениям всех ее полей. Но он и наиболее трудоемок, поскольку при большом количестве полей элемент **Where** получается очень большим. Вариант **upWhere-Changed** требует меньших затрат, но он и менее надежен. Если, пока шло редактирование записи в данной транзакции, другая транзакция изменила в этой записи поля, отличные от измененных данной транзакцией, то фиксироваться результаты будут уже практически для другой записи. Еще менее надежен вариант **upWhereKeyOnly**, хотя он наиболее быстрый. Его можно рекомендовать только в случае, если есть уверенность, что данное приложение — единственное, модифицирующее таблицу на данном отрезке времени.

10.3 InterBase — работа на платформе клиент/сервер

10.3.1 Общие сведения

Все рассмотренное выше по созданию сетевых приложений, работающих с базами данных, относится и к платформе клиент/сервер. Но некоторые особенности этой платформы, в частности, создание обзоров — Views и использование хранимых на сервере процедур, еще не обсуждались. Прежде, чем заняться этим, рассмотрим в качестве примера работу с сервером InterBase. Это тем более оправдано, что в C++Builder имеется локальный сервер InterBase — Borland InterBase Server. Он является усеченной локальной версией Borland Workgroup сервера, который находит применение в больших реальных задачах.

Borland InterBase Server позволяет в локальном варианте разрабатывать программы, которые в дальнейшем будут работать на реальных системах. При этом во время разработки отпадает нужда в реальном отдельном сервере, можно не манипулировать реальными данными, рискуя их испортить, и не решать сложных сетевых проблем. В этом и состоят преимущества локального сервера.

В то же время приложение, отлаженное с использованием Borland InterBase Server, можно затем легко перенести на реальный сервер InterBase, а, учтя некоторые особенности локальных диалектов SQL, можно перенести и на другие имеющиеся на рынке системы, такие, как Informix, Microsoft SQL Server, Oracle, Sybase и др.

10.3.2 Программа Server Manager

Прежде, чем начинать работать с базами данных в Borland InterBase Server, ознакомимся с программой — диспетчером Server Manager. Она понадобится нам, в частности, чтобы зарегистрировать себя как пользователя. Это необходимо, так как создание и использование баз данных в InterBase потребует от нас указания пользователя, создавшего базу данных, и его пароля.

Вызов Server Manager возможен из раздела главного меню C++Builder Tools, если раздел Server Manager туда включен, или непосредственным выполнением файла **Ibmgr32.exe** (он обычно расположен в каталоге ...\\Program Files\\IntrBase Corp\\InterBase\\BIN или в ...\\Program Files\\Borland\\IntrBase\\BIN).

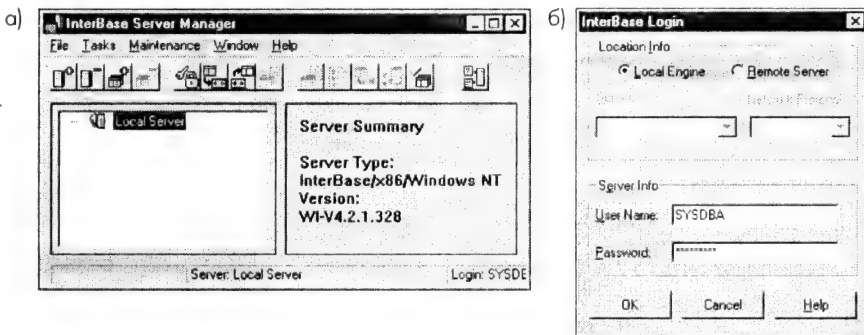
После вызова перед вами откроется окно, представленное на рис. 10.6 а. Точнее сначала панели этого окна будут пустыми.

Прежде всего вам надо соединиться с сервером. Для этого выполните команду File | ServerLogin. При этом вы увидите окно, представленное на рис. 10.6 б. В этом окне должен быть включен индикатор Local Engine — локальный сервер. В окнах редактирования надо ввести имя пользователя (User Name) и пароль (Password). Если вы хотите ввести нового пользователя (в данном случае — себя), то должны соединиться под именем администратора баз данных. Это имя — **SYSDBA**. Пароль адми-

нистратора (если, конечно, он не изменен на вашем компьютере) — **masterkey**. Пароль чувствителен к регистру, так что вводите этот пароль в нижнем регистре.

Щелкните на ОК и окно Server Manager приобретет вид, показанный на рис. 10.6 а.

Рис. 10.6.
Окно
диспетчера
Server Manager
(а) и окно
соединения с
сервером (б).



Теперь попробуйте зарегистрировать себя как пользователя. Заодно, если вы собираетесь пользоваться базой данных InterBase — **ib**, которую вы можете найти на прилагаемом к книге диске, зарегистрируйте и меня: мое имя как пользователя, на которое создана база данных — «А» (латинская буква), а пароль — «1». Кстати, поскольку в дальнейшем система не раз будет запрашивать у вас имя и пароль, то для учебных целей полезно имя делать предельно коротким, а пароль — тоже коротким и цифровым (это избавит вас от необходимости следить за тем, какой язык в данный момент использует Windows — русский или английский).

Чтобы зарегистрировать нового пользователя или изменить информацию о зарегистрированных пользователях, надо выполнить команду **User Security**. Перед вами откроется диалоговое окно, подобное представленному на рис. 10.7 а. Кнопки этого окна позволяют зарегистрировать нового пользователя (кнопка **Add User**), изменить информацию зарегистрированного пользователя, например, его пароль (кнопка **Modify User**), удалить пользователя из списка (кнопка **Delete User**). Сейчас вы хотите ввести нового пользователя. Щелкните на кнопке **Add User** и в открывшемся окне (рис 10.7 б) введите имя пользователя (**User Name**), пароль (**Password**), его подтверждение (**Confirm Password**) — т.е. повторите его еще раз. В нижних окнах редактирования можете (но не обязательно) написать свои имя, отчество и фамилию. Щелкните на ОК и в окне рис. 10.7 а должно появиться ваше имя как пользователя.

Мы рассмотрели те операции с Server Manager, которые нам потребуются для дальнейшего. Остальные возможности Server Manager вам придется изучать самостоятельно по встроенной в программу справке.

10.3.3 Windows ISQL

Программное средство Windows ISQL (Interactive SQL) представляет собой интерфейс для выполнения запросов SQL в интерактивном режиме или из специальных файлов.

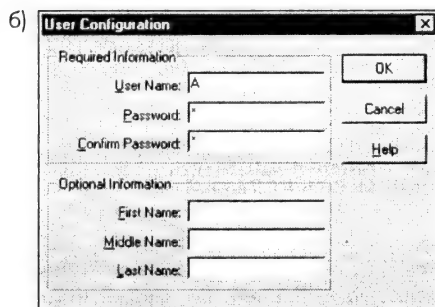
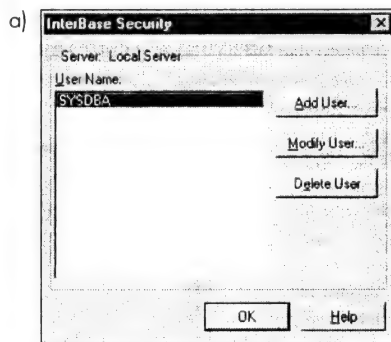
В противоположность Paradox и dBase, которые хранят таблицы в отдельных файлах, InterBase хранит все объекты базы данных в одном файле. Этот файл (а не каталог, как раньше) и является базой данных.

Чтобы создать новый файл базы данных, надо сделать следующее:

- Вызовите WISQL — или из раздела меню **Tools**, если раздел WISQL туда включен, или непосредственно выполните файл **Wisql32.exe** для 32-разрядных версий C++Builder (каталог ...\\Program Files\\InterBase Corp\\InterBase\\BIN).

Рис. 10.7.

Окно данных о пользователях (а) и окно ввода информации о новом пользователе (б)

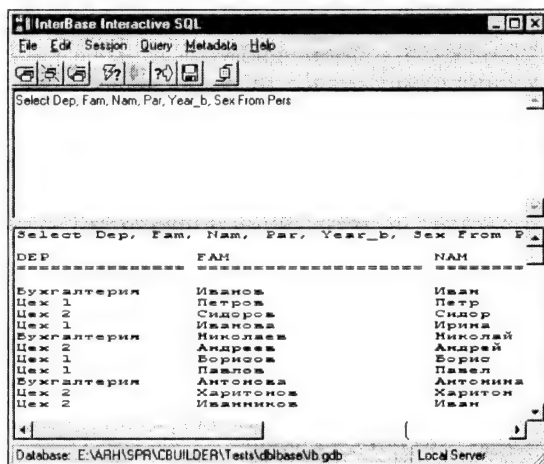


Кстати, если вы намерены и дальше работать с WISQL, целесообразно включить его вызов в меню Tools (если его там нет). Для этого надо выполнить команду Tools | Configure Tools и указать в диалоговом окне место расположения выполняемого файла и имя соответствующего ему раздела меню (например, WISQL).

После вызова WISQL перед вами откроется окно, показанное на рис. 10.8. Точнее, такой вид, как на рисунке, окно приобретет позднее, когда вы создадите базу данных и начнете с ней работать.

Рис. 10.8.

Окно WISQL



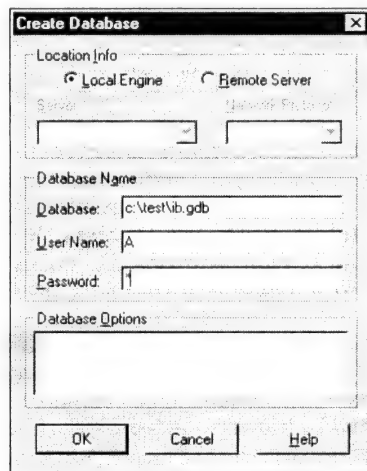
Дальнейшие ваши действия должны сводиться к следующему.

- Выполните команду меню WISQL File | Create Database (вторая быстрая кнопка слева).

- В диалоговом окне (рис. 10.9) напишите имя файла создаваемой базы данных (окно Database) с полным путем. Принятое расширение файлов баз данных в InterBase — **.gdb**. Введите также свое имя пользователя (окно User Name) и пароль (окно Password).
- Щелкните на ОК и база данных окажется созданной и соединенной с вами.

Рис. 10.9.

Окно создания новой базы данных InterBase



Для созданной базы данных полезно установить псевдоним. Как это делать — рассмотрено в разделе 9.3 главы 9. Для базы данных, которую вы можете найти на диске, приложенном к книге, предусмотрен псевдоним **ib**.

После создания базы данных вы в последующие вызовы WISQL можете соединяться с ней с помощью команды **File | Connect to Database** (левая быстрая кнопка на рис. 10.8) В возникающем при этом диалоговом окне вы можете выбрать свою базу данных из выпадающего списка, который содержит сведения о нескольких последних контактах, или написать имя файла с путем заново, если его нет в выпадающем списке.

Теперь перейдем к созданию таблицы. Она создается с помощью рассмотренного ранее оператора **SQL Create Table**. Операторы SQL пишутся в верхней части диалогового окна WISQL (рис. 10.8). Например, упрощенный вариант таблиц базы данных **ib**, с которой мы в дальнейшем будем работать, может быть создан операторами:

```
create table Pers(  
    Num smallint Not Null Primary Key,  
    Dep char(15),  
    Fam char(20) Not Null,  
    Nam char(20) Not Null,  
    Par char(20) Not Null,  
    Year_b smallint DEFAULT 1950,  
    Sex char(1) DEFAULT 'M',  
    Charact blob,  
    Photo blob  
);  
  
create table Dep(  
    Dep char(15) Not Null Primary Key,  
    Proisv char(1)  
);
```

Обратите внимание, что в таблице Pers поле **Sex** (пол) имеет символьный тип размером в 1 символ, хотя для такого поля можно было бы использовать булев

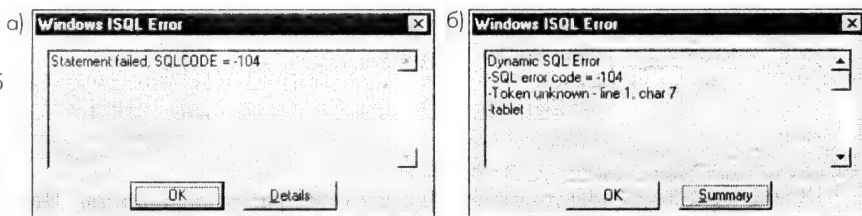
тип. Но в InterBase нет булева типа, так что приходится пользоваться вместо него символьным.

Более полный вариант создания таблиц базы данных **ib** вы можете посмотреть на прилагаемом к книге диске. Подробное описание оператора **Create Table**, используемого в этом полном варианте, вы можете найти в книге [8].

После того, как вы написали оператор создания таблицы, выполните команду Query | Execute (четвертая слева быстрая кнопка на рис. 10.8). Если в вашем операторе SQL нет ошибок, то в нижней части окна появятся результаты выполнения данного запроса. Если же в операторе обнаружались синтаксические ошибки, то появится окно с сообщением об ошибке (рис. 10.10). Оно, конечно, ничего вам не скажет кроме того, что была какая-то ошибка. Но не спешите щелкать на ОК и искать сделанную ошибку. Щелкните лучше на кнопке Details (детали) и вы увидите окно, которое может выглядеть так, как показано на рис. 10.10 б. В данном случае в приведенном выше операторе была преднамеренно сделана ошибка: вместо «table» написано «tablet». Как видите, ошибка определена с точностью до символа: line 1, char 7 (строка 1, символ 7) и приведено нераспознанное слово — tablet. Так что просмотр детальных пояснений ошибок серьезно поможет вам в отладке.

Рис. 10.10.

Окна WISQL с сообщением об ошибке (а) и с детализацией ошибки (б)



Если вы создали таблицу, можете попробовать ее заполнить с помощью операторов **Insert Into** языка SQL. Например:

```
Insert Into PERS(Num, Dep, Fam, Nam, Par, Year_b, Sex)
Values(1, "Бухгалтерия", "Иванов", "Иван", "Иванович",
1950, "м")
```

Выполнив этот оператор, вы создадите первую запись в таблице. Конечно, писать для каждой записи такой длинный оператор — задача неблагодарная. К счастью, этого делать не надо. Выполнив один оператор, выберите команду Query | Previous (пятая слева быстрая кнопка на рис. 10.8). Перед вами появится предыдущий оператор и вам останется только заменить в нем содержание текстов в списке **Values**.

В подобном режиме вы можете выполнять любые запросы SQL: **Select**, **Update**, **Delete**, **Set transaction**, **Commit** и др.

При выполнении оператора выбора **Select** в нижнем окне вы увидите результат выполнения. Можете вычислять совокупные характеристики, использовать объединения и т.п.

При расчетах можно устанавливать ряд опций оператором

```
SET <опция>;
```

Отмена опции осуществляется оператором:

```
SET <опция> OFF;
```

Содержимое нижнего окна с результатами вы можете сохранить для последующего использования в текстовом файле, щелкнув для этого на кнопке **Save Results** и указав в диалоге имя файла.

Когда вы завершаете работу с WISQL после каких-то изменений в базе данных, программа задает вам вопрос: «Commit work for database?». Дело в том, что в процессе работы, если вы не использовали оператор **Commit**, все изменения прово-

дились во временной копии базы данных. И теперь вам задается вопрос: «Зафиксировать результаты в базе данных?». При положительном ответе данные будут зафиксированы, а при отрицательном — нет.

Мы рассмотрели один из вариантов работы с WISQL — интерактивный, при котором в верхнем окне вводятся операторы SQL, кнопкой *Execute query* они выполняются, а в нижнем окне можно просмотреть результаты. Имеется и другой вариант — работа из файла запросов, который называется *script*-файлом. Это обычный текстовый файл с расширением *.sql* по умолчанию. Запуск выполнения запросов из файла осуществляется командой *File | Run an ISQL Script*.

Поскольку прежде всего надо связаться с интересующей вас базой данных (в интерактивном режиме вы выполняли для этого команду меню *File | Connect to Database*), файл запросов обычно начинается с команды запроса соединения, которая имеет следующий синтаксис:

```
Connect <база данных> User <имя пользователя> Password <пароль>;
```

Например:

```
CONNECT "c:\test\ib.gdb" USER "A" PASSWORD "1";
```

Затем в файле могут размещаться любые запросы SQL. Можно вводить в текст комментарию в стиле *C: /* <комментарий> */*.

Пример подобного файла вы можете найти на прилагаемом к книге диске. Этот файл *createdb.sql* создает базу данных *ib.gdb*, создает и заполняет ее таблицы, создает в ней обзоры и хранимые процедуры (об этих возможностях будет рассказано в следующих разделах).

В заключение коротко остановимся на некоторых командах меню WISQL. Меню *File* имеет раздел *Drop Database*, позволяющий уничтожить базу данных, с которой в данный момент поддерживается связь. Раздел *Save Result to a File* позволяет сохранить в текстовом файле данные, полученные в окне результатов, а раздел *Save Session to a File* позволяет сохранить и сами запросы, и их результаты.

Меню *Session* позволяет установить различные опции, определяющие состав и форму представления результатов в нижнем окне программы. Меню *Metadata* содержит раздел *Show* — просмотра метаданных базы данных. Метаданные отображают структуру компонентов, входящих в базу данных. Однако раздел *Show* позволяет только просмотреть список соответствующих таблиц, процедур и т.д. Более полезны разделы *Extract Database*, *Extract Tables*, *Extract View*, позволяющие извлекать метаданные — операторы SQL, с помощью которых была создана база данных или отдельные ее компоненты.

10.3.4 Обзоры — Views

Обзоры (*views*) — это дочерние образования таблицы, в которые помещается некоторое подмножество записей, содержащих все, или только указанные поля. Пусть, например, в некоторой организации имеется таблица персонала, подобная используемой нами таблице *Pers*. Из нее имеет смысл создать обзоры сотрудников, работающих в каждом подразделении, и раздать их руководителям подразделений. Тогда каждый руководитель будет иметь базу данных своих сотрудников, но не будет иметь доступа к сведениям о сотрудниках других отделов. А общая, базовая таблица имеется, например, у руководства предприятия. При этом можно сохранять конфиденциальность — в каждом отделе будут сведения только о своих сотрудниках, а какая-то конфиденциальная информация (какие-то поля) могут вообще не включаться в обзоры, а храниться только в общей таблице под паролем и быть доступными только избранным представителям администрации. Все обзоры и базовая таблица, на основе которой они созданы, связаны друг с другом. Если в таблице или в каком-то обзоре проведены изменения (поступил на работу новый сотрудник, кто-то уволился, у кого-то изменился адрес, телефон, семейное положение), все они тут же отразятся во всех обзорах, в которых есть соответствующие записи, и в базовой таблице.

Синтаксис оператор создания обзора следующий:

```
CREATE VIEW <имя обзора> AS SELECT <список полей>
FROM <таблица> WHERE <условие>
```

В нашем примере этот оператор может иметь вид:

```
CREATE VIEW DEP_1 AS SELECT Fam, Nam, Par, Year_b, Sex
FROM Pers WHERE Dep = 'Цех 1'
```

Уничтожение ранее созданного обзора осуществляется оператором:

```
DROP VIEW <имя обзора>
```

Например, уничтожить обзор, созданный в приведенном выше примере, можно оператором:

```
DROP VIEW Dep_1
```

10.3.5 Хранимые на сервере процедуры

10.3.5.1 Создание выполняемых процедур

Клиенты, связанные с SQL сервером, могут обращаться к его мощностям для решения сложных задач. Это эффективнее, чем решать подобные задачи на более слабом компьютере клиента. К тому же, это позволяет минимизировать объемы информации, пересылаемой через сеть. Например, если пользователю нужно отыскать только одну конкретную запись и просмотреть в ней некоторые поля, целесообразно всю процедуру поиска выполнить на сервере, а пользователю переслать только интересующие его поля найденной записи. В идеале клиент должен получать только тот минимум данных, которыми манипулирует пользователь.

Многие серверы, в том числе InterBase, разрешают написание специальных процедур, хранимых и выполняемых на сервере. К сожалению, язык написания хранимых процедур различен в разных системах. И язык InterBase имеет мало общего с языком Sybase или Microsoft SQL сервера.

InterBase поддерживает 2 вида хранимых процедур: *выполняемые (Execute)*, которые могут передавать параметры и которые манипулируют данными, и *процедуры выбора (Select)*, которые представляют собой таблицы только для чтения, но которые воспринимают параметры, определяющие возвращаемые результаты.

Рассмотрим сначала процесс создания выполняемых хранимых процедур. Простейший путь для этого — написание соответствующих script-файлов и их выполнение с помощью WISQL. Следует подчеркнуть, что использование интерактивного режима работы с WISQL для создания хранимых процедур не допускается.

Структура соответствующего файла может иметь вид:

```
CONNECT <база данных> USER <имя пользователя>
PASSWORD <пароль>;
SET AUTODLL OFF;
SET TERM ^ ;
```

```
CREATE PROCEDURE <имя процедуры> AS
BEGIN
.....
END^
SET TERM ; ^
COMMIT;
```

Первая команда **Connect** связывается с базой данных. Вторая команда **Set** отключает опцию **AUTODDL**. Это предотвращает процедуру от преждевременного создания. Следующая команда **Set Term** заменяет символ окончания «;» на «^». Зачем это надо? Операторы InterBase и операторы SQL, используемые при написании текста процедуры, имеют одинаковые символы окончания — «;». Поэтому

транслятор не сможет разобраться, где операторы InterBase, а где операторы процедуры. Чтобы не возникало этой путаницы, команда **Set** временно заменяет символ окончания операторов InterBase на «^». В результате транслятор будет знать, что если встретится оператор, кончающийся символом «^» — это будет оператор InterBase. А символы окончания операторов процедуры остались прежними — «;».

После рассмотренных подготовительных операторов следует оператор создания процедуры **Create Procedure**, в котором указывается имя процедуры. После ключевого слова **AS** следуют операторы самой процедуры, начинающиеся с **begin** и кончающиеся **end^**. После этого оператора транслятор начинает обрабатывать дальнейшее как операторы InterBase. Следующий оператор **Set Term** переключает символ окончания на традиционный — «;», а оператор **Commit** фиксирует результат создания процедуры в базе данных.

Файл может содержать операторы создания нескольких процедур. Каждая из них компилируется отдельно. Те, которые откомпилировались без синтаксических ошибок, будут записаны в базу данных.

После того, как процедура создана, ее можно тестировать в WISQL в интерактивном режиме. Для вызова процедуры достаточно выполнить оператор

```
EXECUTE PROCEDURE <имя>;
```

Процедуры могут выполняться любым путем: интерактивно в WISQL, из приложения, могут вызываться из других процедур и даже из самих себя — т.е. предусмотрена возможность рекурсии.

Удалить из базы данных процедуру, которая оказалась ошибочной, можно оператором

```
DROP PROCEDURE <имя>;
```

Рассмотренная выше структура процедуры — простейшая. Она соответствует процедуре, в которую не передается никаких параметров и которая ничего не возвращает, а просто, например, изменяет базу данных. Рассмотрим теперь пример более сложной процедуры с параметрами. В эту процедуру с именем **GetInf** передаются фамилия, имя и отчество сотрудника. Процедура возвращает информацию о сотруднике: год рождения, подразделение, в котором сотрудник работает, и пол. Если запись сотрудника в базе данных не обнаружена, то в качестве года рождения возвращается 0, что может являться для приложения, вызвавшего процедуру, сигналом отсутствия затребованной записи.

Без учета стандартных начальных и конечных операторов создание такой процедуры осуществляется следующим кодом:

```
CREATE PROCEDURE GetInf
    (pFam char(20), pNam char(20), pPar char(20))
    RETURNS (pYear integer, pDep char(15), pSex char(1))
AS
BEGIN
    pYear=0;
    SELECT Year_b, Dep, Sex From Pers
    WHERE (Fam=:pFam) and (Nam=:pNam) and (Par=:pPar)
    INTO pYear, pDep, pSex;
END;^
```

Просмотрев данный текст, вы можете увидеть, что передаваемые в процедуру параметры определяются через прискобочную запись, следующую за именем процедуры, причем для каждого параметра указывается его тип. Это очень похоже на объявление процедуры в любом алгоритмическом языке. Возвращаемые параметры указываются аналогичным образом после ключевого слова **RETURNS**. В приведенной процедуре все имена начинаются с символа «р», но это, конечно, не обязательно. Просто подобное обозначение позволяет легче читать текст процедуры и не путать параметры с именами полей.

Тело процедуры состоит всего из двух операторов. Первый из них задает значение возвращаемого параметра **pYear**, равное нулю. Второй оператор **Select** ищет запись, в которой значения полей **Fam**, **Nam** и **Par** совпадают соответственно с заданными значениями фамилии, имени и отчества (параметрами **pFam**, **pNam** и **pPar**). Если такая запись нашлась, то в параметры **pYear**, **pDep** и **pSex** (указаны после ключевого слова **Into**) передаются значения полей **Year_b**, **Dep** и **Sex**, указанных после ключевого слова **Select**. Таким образом конструкция **Select ... Into** позволяет заносить результаты отбора в параметры. Если искомая запись отсутствует в базе данных, оператор **Select** ничего в параметры не заносит. Следовательно, в возвращаемом параметре **pYear** останется 0, присвоенный предыдущим оператором. Это будет сигналом для программы, вызвавшей данную процедуру, об отсутствии записи.

Вызов такой процедуры из WISQL может осуществляться оператором вида:

```
EXECUTE PROCEDURE GETINF("Иванов", "Иван", "Иванович")
```

Приведенная процедура иллюстрирует некоторые возможности описания выполняемых хранимых процедур. В процедурах можно использовать циклы, операторы условной передачи информации и многое другое. В подробностях это все можно посмотреть во встроенной в WISQL справке.

В процедурах можно вводить внутренние локальные переменные. Они вводятся конструкцией вида

```
DECLARE VARIABLE <список переменных и их типов>
```

помещаемой после ключевого слова **as**.

10.3.5.2 Вызов выполняемых хранимых процедур из приложения

Для вызова выполняемых хранимых на сервере процедур в библиотеке C++Builder на странице Data Access имеется компонент **StoredProc**. Он позволяет передавать информацию в хранимые процедуры и воспринимать возвращаемую информацию с помощью параметров.

Перенеся на форму компонент **StoredProc**, надо прежде всего установить его свойство **DatabaseName**, выбрав из списка псевдоним базы данных, с которой вы хотите работать. Затем можно установить свойство **StoredProcName**. В выпадающем списке этого свойства перечислены хранимые процедуры, определенные в базе данных.

Входные параметры процедуры заносятся в процессе работы в свойство **Params**. Имена параметров, содержащихся в этом свойстве, вы можете узнать, посмотрев это свойство с помощью Инспектора Объектов. При этом вы увидите то же окно установки атрибутов параметров, которое ранее было рассмотрено в компоненте **Query** (рис. 10.1). Остановимся только на одном свойстве, которое ранее не рассматривалось — **ParamType**. Это свойство — тип параметра, указывающий на способ его использования, может принимать значения:

ptUnknown	Тип неизвестен или не определен. В этом случае перед выполнением процедуры необходимо установить этому параметру другой, определенный тип
ptInput	Входной параметр, передаваемый в процедуру
ptOutput	Выходной параметр, возвращаемый из процедуры
ptInputOutput	Параметр может использоваться и как входной, и как выходной.
ptResult	Параметр используется как возвращаемая величина. Обычно применяется для возвращения сообщений об ошибках или о режиме завершения. Хранимая процедура может иметь только один параметр такого типа

Если вы связываете компонент **StoredProc** с конкретной процедурой во время проектирования, то свойства **ParamType** всех параметров устанавливаются автоматически в соответствии с описанием процедуры. Так что вам не приходится их устанавливать вручную.

Перед вызовом процедуры приложение должно занести в **Params** значения параметров. Свойство **ParamBindMode** компонента **StoredProc** задает способ программного доступа к параметрам. Это свойство может принимать значения: **pbByName** — доступ по имени и **pbByNumber** — доступ по индексу. Как правило, используется доступ по имени методом **ParamByName**. Например, **StoredProc1->ParamByName("PFAM")** — значение параметра **PFAM**. В редких случаях, когда имена параметров неизвестны, можно использовать доступ по индексу. Последовательность параметров в свойстве **Params** определяется последовательностью их определения в объявлении процедуры. Например, **StoredProc1->Params->Items[0]** — значение первого параметра.

Вызов процедуры осуществляется методом **ExecProc**. Возвращаемые процедурой значения ее выходных параметров могут быть взяты из свойства **Params**. Они располагаются там после входных параметров. При получении результатов может быть использован метод **ParamByName** или доступ по индексу.

Постройте пример, использующий рассмотренную ранее процедуру **GetInf**. Введите в приложение окна редактирования **FamEdit**, **NamEdit** и **ParEdit**, в которых пользователь может задавать фамилию, имя и отчество сотрудника, информация о котором его интересует. Введите также метки **LDep**, **LYear** и **LSex**, в которых будет отображаться возвращаемая процедурой информация. И введите кнопку **Выполнить**, при щелчке на которой должен выполняться код:

```
StoredProc1->ParamByName("PFAM")->AsString = FamEdit->Text;
StoredProc1->ParamByName("PNAM")->AsString = NamEdit->Text;
StoredProc1->ParamByName("PPAR")->AsString = ParEdit->Text;
StoredProc1->ExecProc();
if (StoredProc1->ParamByName("PYEAR")->AsInteger == 0)
    Application->MessageBox("В базе данных запись отсутствует",
        "Ошибка", MB_OK);
else
{
    LDep->Caption = StoredProc1->ParamByName("PDEP")->AsString;
    LYear->Caption = StoredProc1->ParamByName("PYEAR")->AsString;
    LSex->Caption = StoredProc1->ParamByName("PSEX")->AsString;
}
```

Приведенный код, вероятно, в дополнительных комментариях не нуждается. Запустите ваше приложение и попробуйте с его помощью получить информацию о сотруднике, о котором в базе данных имеется запись, и о сотруднике, отсутствующем в базе данных. Для контроля полезно добавить к приложению контрольную таблицу, которая отображала бы содержимое базы данных.

На диске вы можете найти более развернутый пример, работающий с процедурами базы данных **ib**.

10.3.5.3 Хранимые процедуры выбора

Рассмотренные ранее выполняемые процедуры возвращали параметры. В отличие от них процедуры выбора возвращают множество результатов, таблицы.

Как пример, рассмотрим процедуру выбора с именем **Selectd**, возвращающую информацию о сотрудниках, работающих в указанном отделе.

```
CREATE PROCEDURE Selectd (pDep char(15))
RETURNS (pFam char(20), pNam char(20),
        pPar char(20), pYear integer, PSex char(1))
AS
BEGIN
```

```

FOR SELECT Fam, Nam, Par, Year_b, Sex From Pers
WHERE (Dep=:pDep)
INTO pFam, pNam, pPar, pYear, pSex
DO
  SUSPEND;
END; ^

```

Заголовок оператора **Create Procedure** выглядит так же, как и для выполняемой процедуры. В нем определяется передаваемый параметр (в данном случае **pDep**) и возвращаемые параметры, перечисляемые в списке после ключевого слова **Returns**. Так же, как в выполняемой процедуре, используется блок **begin...end**, хотя в процедурах выбора в нем обычно имеется только один оператор **For Select**. Этот оператор осуществляет выбор, как и оператор **Select ... Into** в выполняемых процедурах, но отличается от него следующими особенностями. Он обрабатывает не сразу всю таблицу, а по одной записи за раз, причем имеется блок **Do**, обрабатывающий каждую возвращенную запись. В данном примере (и обычно) этот блок использует ключевое слово **SUSPEND**, означающее возвращение записи в предложении **SQL**, вызвавшее данную процедуру.

Для использования хранимой на сервере процедуры выбора не требуется никакой специальной команды, как для выполняемых процедур. Вызов осуществляется обычной командой **Select**, в которой имя процедуры выбора фигурирует как обычная таблица. Только в данном случае для этой таблицы могут указываться параметры. Например, из **WISQL** вызов приведенной процедуры имеет вид:

```
Select * from Selectd ("Бухгалтерия")
```

В этот оператор **Select** можно включать вычисления совокупных характеристик и вообще работать с процедурой выбора как с обычной таблицей.

Из приложения **C++Builder** процедуру выбора можно вызывать элементом **Query**, не используя **StoredProc**, с помощью аналогичного оператора **Select**. Впрочем, можно использовать и **StoredProc**. Это, однако, менее удобно, так как при этом нельзя включать результат в предложения **SQL**.

10.4 Доступ к базам данных через Microsoft ActiveX Data Objects (ADO)

10.4.1 Соотношение между компонентами BDE и ADO

В **C++Builder 5** появилась возможность работы с базами данных посредством разработанной в **Microsoft** технологии **ActiveX Data Objects (ADO)**. **ADO** — это пользовательский интерфейс к любым типам данных, включая реляционные и не реляционные базы данных, электронную почту, системные, текстовые и графические файлы. Связь с данными осуществляется посредством так называемой технологии **OLE DB**.

Использование **ADO** является альтернативой **Borland Database Engine (BDE)**, обеспечивающей более эффективную работу с данными. Для использования этой возможности на вашем компьютере должна быть установлена система **ADO 2.1** или более старшая версия. Кроме того должна быть установлена клиентская система доступа к данным, например, **Microsoft SQL Server**, а в **ODBC** должен иметься драйвер **OLE DB** для того типа баз данных, с которым вы работаете.

Для работы с **ADO** в **C++Builder 5** предусмотрены компоненты, расположенные на новой странице библиотеки — **ADO**. Они инкапсулируют такие объекты **ADO**, как **Connection**, **Command** и **Recordset**. Это обеспечивается соответственно новыми компонентами **C++Builder ADOConnection**, **ADOCommand** и **ADODataset**.

Связь с базой данных в технологии ADO осуществляется обычной цепочкой: набор данных \Rightarrow источник данных (компонент **DataSource**) \Rightarrow компоненты управления и отображения данных (**DBGrid**, **DBEdit** и др.). Отличие заключается только в первом звене этой цепочки, в котором вместо компонентов, расположенных на странице Data Access библиотеки используются компоненты, расположенные на странице ADO.

Большинство компонентов, предназначенных для работы с ADO, аналогичны прежним компонентам, работающим с BDE:

Компонент ADO	Компонент BDE
ADOTable	Table
ADOQuery	Query
ADOSToredProc	StoredProc
ADOConnection	Database
ADODataset	Table, Query, StoredProc
ADOCommand	-
RDSConnection	-

Ниже приведена краткая характеристика основных компонентов ADO.

ADOConnection	Используется для связи с набором данных ADO. Может работать с несколькими компонентами наборов данных как диспетчер выполнения их команд
ADODataset	Универсальный компонент связи с наборами данных, который может работать в различных режимах, заменяя связанные с BDE компоненты Table , Query , StoredProc . Может связываться с одной или множеством таблиц. Связь осуществляется непосредственно, или через ADOConnection
ADOTable	Используется для работы с одной таблицей. Может связываться с ней непосредственно, или через ADOConnection
ADOQuery	Используется для работы с набором данных с помощью запросов SQL, включая такие запросы языка DDL (data definition language), как CREATE TABLE . Может связываться с набором данных непосредственно, или через ADOConnection
ADOSToredProc	Используется для выполнения процедур, хранимых на сервере. Может связываться с набором данных непосредственно, или через ADOConnection
ADOCommand	Используется в основном для выполнения команд SQL, не возвращающих множество результатов. Может также совместно с другими компонентами использоваться для работы с таблицами. Может связываться с набором данных непосредственно, или через ADOConnection

В качестве общей характеристики можно сказать, что в некоторых отношениях компоненты ADO мощнее компонентов BDE, но в то же время ряд возможностей компонентов BDE в них не реализован. Например, они не могут использовать словари свойств, что приводит к лишней работе при создании приложений. Не все

источники данных ADO могут работать со всеми типами полей. Например, источник данных Paradox ADO не работает с графикой. Так что надо серьезно рассматривать все за и против, прежде чем переходить от BDE к ADO.

10.4.2 Задание соединения компонентов ADO с базой данных

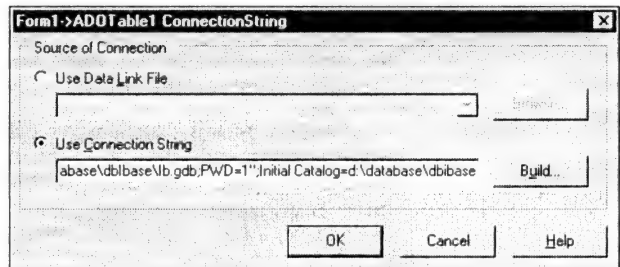
В отличие от компонентов BDE — **Table**, **Query** и других, в компонентах ADO нет свойства **DatabaseName**, указывающего базу данных. Доступ к базе данных осуществляется или с помощью строки соединения — свойства **ConnectionString**, или с помощью отдельного компонента **ADOConnection**, имя которого задается в свойстве **Connection** других компонентов.

Рассмотрим соединение с базой данных с помощью свойства **ConnectionString** на примере компонента **ADOTable**. Свойство **ConnectionString** представляет собой строку, содержащую параметры соединения. Отдельные параметры отделяются друг от друга точками с запятой. В C++Builder предусмотрено специальное диалоговое окно, облегчающее работу по формированию строки соединения.

Перенесите на форму компонент **ADOTable** и в Инспекторе Объектов нажмите кнопку с многоточием около свойства **ConnectionString**. Перед вами откроется окно, показанное на рис. 10.11. Верхняя радиокнопка **Use Data Link File** позволяет использовать файл связи **.udl**. На этих файлах, используемых в Windows, мы останавливаться не будем. Вы можете найти материал о них в книгах [7] и [8]. Нижняя радиокнопка **Use Connection String** позволяет сформировать строку соединения в режиме диалога. Включите эту радиокнопку и нажмите кнопку **Build** (Сформировать).

Рис. 10.11.

Первое диалоговое окно задания строки соединения



Перед вами откроется многостраничное окно задания свойств соединения. На странице **Provider** вы должны указать провайдер OLE DB, который собираетесь использовать для доступа к данным. Во многих случаях вас устроит выбор **Microsoft OLE DB Provider for ODBC Drivers**. Однако, например, для работы с **Microsoft SQL Server** или **Oracle** надо выбрать другие разделы списка.

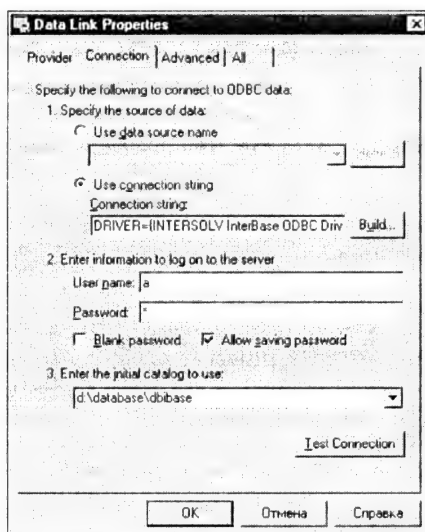
Выбрав провайдер, вы должны перейти на страницу **Connection** (рис. 10.12). Впрочем, если вы на странице **Provider** нажмете кнопку **Next**, то этот переход свершится автоматически.

На странице **Connection** вы должны указать, как вы будете соединяться с ODBC. Выбрав кнопку **Use data source name**, вы можете задать из выпадающего списка имя источника данных (**data source name** — **DSN**), зарегистрированного в ODBC. Кнопка **Use connection string** позволяет вам задать строку соединения самостоятельно, не прибегая к зарегистрированным DSN.

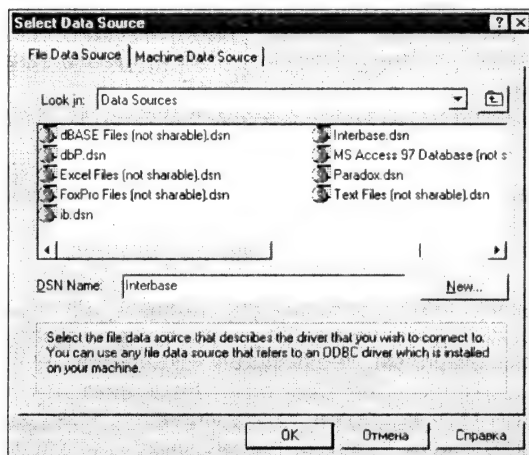
Выберите кнопку **Use connection string** и нажмите **Build**. Перед вами откроется окно, представленное на рис. 10.13. В нем вы можете выбрать один из зарегистрированных в ODBC файлов источников данных. Если нужного вам файла источника нет в списке, вы можете добавить его, используя кнопку **New**. Например, первоначально на вашем компьютере может не быть источников данных **ib**, **Intehbase**, **Paradox**, показанных на рис. 10.13. Тогда, чтобы создать новый файл источника, нажмите кнопку **New**.

Рис. 10.12.

Страница Connection основного окна задания свойств соединения

**Рис. 10.13.**

Выбор источника данных



Перед вами пройдет череда диалоговых окон, в которых вы должны указать характеристики создаваемого источника данных. Если вы создаете источник данных InterBase, вам надо указать в них драйвер INTERSQL InterBase ODBC Driver (*.gdb). При создании источника данных Paradox выберите драйвер Microsoft Paradox Driver (*.dbf). После этого вам надо задать произвольное имя нового источника данных. Далее вам потребуется указать базу данных, для которой вы создаете источник, и такие характеристики, как имя пользователя, пароль и т.п. После всего этого вы опять увидите окно рис. 10.13, но в нем уже будет созданный вами источник данных.

После того, как в окне рис. 10.13 вы выбрали существовавший ранее или созданный новый файл источника данных, вы вернетесь в основное окно задания свойств соединения на страницу Connection (рис. 10.12). Если вы щелкнете на OK, то вернетесь в окно рис. 10.11, в котором в нижнем окошке будет записана сформированная вами строка соединения. Для InterBase она может иметь вид:

```
Provider=MSDASQL.1;Password=1;Persist Security Info=True;
User ID=a;Mode=Read|Write;Connect Timeout=15;Extended
Properties="DRIVER={INTERSQL InterBase ODBC Driver
```

```
(* .gdb) };UID=A;DB=D:\Database\dbIbase\Ib.gdb;PWD=1";Locale
Identifier=1049;Initial Catalog=D:\Database\dbIbase
```

Для Paradox сформированная строка соединения может иметь вид

```
Provider=MSDASQL.1; Persist Security Info=False; Extended
Properties="CollatingSequence=ASCII; DBQ=D:\DATABASE\BPAR;
DefaultDir=D:\DATABASE\BPAR; Driver={Microsoft Paradox Driver
(*.db) }; DriverId=538; FIL=Paradox 5.X; FILEDSN=C:\Program
Files\Common Files\ODBC\Data Sources\Paradox.dsn; MaxBufferSize=2048;
MaxScanRows=8; PageTimeout=5; ParadoxNetPath=C:\WINDOWS\SYSTEM;
ParadoxNetStyle=3.x; ParadoxUserName=admin; SafeTransactions=0;
Threads=3; UID=admin; UserCommitSync=Yes;"
```

Как видите, в результате не слишком сложного диалога сформировались строки, которые иным способом записать было бы весьма трудно.

В окне рис. 10.12 вы можете также занести дополнительную информацию: имя пользователя (User name), пароль доступа (Password), ввести начальный каталог (Enter the initial catalog to use). Введенное вами имя пользователя занесется в строку соединения в виде параметра (см. приведенные выше текст для **ib**), например:

```
User ID=a;
```

В дальнейшем это имя (в данном примере — «a») будет отображаться в окне запроса пароля при соединении с базой данных. Но пароль сам по себе в строку соединения не занесется, если только вы не включите индикатор Allow saving password. Тогда в строку соединения занесется соответствующий параметр, например:

```
Password=1;
```

В этом случае в дальнейшем при соединениях пароль может не запрашиваться, а если и будет запрошен, то пользователь сможет его не вводить. Но учтите, что тем самым вы раскроете свой пароль для пользователя, который имеет доступ к вашим исходным файлам C++Builder.

Включение индикатора Blank password также разрешит пользователю при запросе пароля во время соединения с базой данных не вводить пароль. Но при этом, если пароль не запомнен в строке соединения, то с базой данных, защищенной этим паролем, соединение установить не удастся.

Кнопка Test Connection позволяет вам проверить правильность всей информации. При нажатии на эту кнопку происходит соединение с базой данных. Если все нормально, будет показано окно с надписью: «Test connection succeeded», свидетельствующей о нормально произведенном тестировании соединения. Если в сформированной строке соединения что-то неправильно, вам будут выданы сообщения об ошибках.

После завершения формирования строки соединения, вы можете перейти в окне рис. 10.12 на страницу Advanced и задать режимы работы с сетью. Страница All сообщает итоговую информацию о соединении и позволяет ее отредактировать.

После того, как вы завершили все операции по формированию строки соединения, нажмите ОК и сформированная строка появится в свойстве **ConnectionString** компонента.

Описанные выше операции формирования строки соединения достаточно громоздки. Хотелось бы, проведя их один раз, запомнить и в дальнейшем использовать при задании соединения других компонентов. К тому же, при смене каталога, в котором расположена база данных, хотелось бы не повторять опять корректировку строки соединения всех приложений, использующих эти данные. Подобное облегчение своей последующей работы можно сделать двумя способами: создать требуемый вам источник данных или создать файл соединения.

Но эти возможности из-за ограничения объема книги вам придется изучить самостоятельно по встроенной справке C++Builder 5 или посмотреть в книгах [7] и [8].

10.4.3 Соединение с помощью компонента **ADOConnection**, управление транзакциями

Соединение компонентов с базами данных можно осуществлять не только через свойство **ConnectionString**, как описано в предыдущем разделе, но и через свойство **Connection**, связывающее данный компонент с компонентом **ADOConnection**. В этом компоненте **ADOConnection**, осуществляющем диспетчеризацию работы с набором данных, соединение задается свойством **ConnectionString**. А во всех прочих компонентах наборов данных достаточно установить в свойстве **Connection** имя компонента **ADOConnection**.

Соединение с базой данных компонентов наборов данных, связанных с **ADOConnection**, происходит, даже если в самом **ADOConnection** не предпринимается никаких действий для открытия базы данных. Достаточно в компоненте набора данных установить свойство **Active = true**, и он свяжется с набором данных. При этом свойство **Connected** компонента **ADOConnection**, показывающее наличие соединения, автоматически установится в **true**. Тогда встает вопрос: «Зачем же нужен компонент **ADOConnection**?».

Компонент **ADOConnection** позволяет управлять атрибутами и условиями соединения подключенных к нему компонентов наборов данных. Свойства **ADOConnection** дают возможность задавать схему блокировки записей, тип курсора, уровень изоляции и многое другое. Методы **ADOConnection** обеспечивают управление транзакциями. Именно из-за этих особенностей и используется **ADOConnection**.

Во время выполнения соединения **ADOConnection** с базой данных осуществляется методом **Open**:

```
HIDEBASE void __fastcall Open(const WideString UserID,  
                               const WideString Password);
```

Например:

```
ADOConnection1->Open("A","1");
```

Параметры **UserID** — идентификатор пользователя и **Password** — пароль не обязательны. Они, как было описано в разделе 10.4.2, могут быть заданы в строке соединения — свойстве **ConnectionString**. Эту информацию приложение может почерпнуть также из диалогового окна, предъявляемого пользователю при соединении с базой данных. Это окно отображается, если установить свойство **LoginPrompt** компонента **ADOConnection** в **true**. Если используется это значение **LoginPrompt**, то имя пользователя и пароль можно не задавать ни в строке соединения, ни в вызове **Open**.

Закрывается соединение с базой данных методом **Close**. Свойство **KeepConnection** определяет, сохраняется ли соединение с базой данных, даже если база данных не открыта. Установка **KeepConnection** в **true** (по умолчанию) сокращает затраты времени и загрузку сети при соединениях с базой данных, но увеличивает затраты ресурсов компьютера.

Существует также метод **CloseDataSets**, закрывающий соединение всех компонентов наборов данных, соединенных с данным компонентом **ADOConnection**, но не закрывающий соединение самого **ADOConnection**.

Альтернативным способом установления и разрыва соединения с базой данных является установка соответственно в **true** или **false** свойства **Connected**. Это свойство можно также использовать для проверки успешности соединения. Если значение **Connected** равно **true**, значит соединение активно. Если значение **Connected** равно **false** и **KeepConnection** также равно **false**, то соединение неактивно.

Свойство **Connected** компонента **ADOConnection** связано со свойствами **Active** компонентов наборов данных, подключенных к данному **ADOConnection**. Если **Connected = false**, а в одном из подключенных компонентов набора данных свойст-

во **Active** переключается в **true**, то **Connected** автоматически устанавливается в **true**. Если же **ADOConnection** разрывает соединение методом **Close** или установкой **Connected** в **false**, то свойства **Active** всех подсоединенных компонентов сбрасывается в **false**. Этим и ограничивается связь свойств **Connected** и **Active**. Если после разрыва соединения компонент **ADOConnection** снова устанавливает соединение методом **Open** или заданием **Connected = true**, то свойства **Active** подсоединенных компонентов остаются равными **false**. Следовательно эти компоненты не подключаются к базе данных. В этом случае надо принять специальные меры для их подключения.

Для этого можно воспользоваться свойством **DataSets** компонента **ADOConnection**, которое является массивом всех компонентов наборов данных, подсоединенных к данному компоненту **ADOConnection**. Количество таких объектов определяется свойством **DataSetCount**. Эти свойства можно использовать, чтобы управлять всеми подсоединенными компонентами наборов данных. Например, код:

```
for(int i = 0; i < ADOConnection1->DataSetCount; i++)  
    ADOConnection1->DataSets[i]->Active = true;
```

обеспечивает активацию соединения с базой данных всех компонентов наборов данных.

Свойство **CursorLocation** определяет, какую библиотеку использует курсор при соединении с базой данных: клиентскую или сервера. Значение **clUserClient** (по умолчанию) обеспечивает большую гибкость. Все данные располагаются на компьютере — клиенте и тут же обрабатываются. При этом возможны операции сортировки и другие, которые могут не поддерживаться сервером. Впрочем, предложения SQL и в этом случае выполняются на сервере и на компьютер клиента передаются уже отобранные данные, соответствующие условию **WHERE** предложения SQL.

Значение свойства **CursorLocation**, равное **clUseServer**, желательно использовать в командах, возвращающих большой объем данных. В подобных случаях использование клиентского курсора может потребовать недопустимых затрат дискового пространства клиента.

Теперь рассмотрим управление транзакциями. Транзакция начинается методом **BeginTrans**, который возвращает целое число, показывающее уровень вложенности данной транзакции. Успешное выполнение **BeginTrans** приводит к генерации события **OnBeginTransComplete** и установке свойства **InTransaction** в **true**. По значению этого свойства можно определить, находится ли компонент в состоянии формирования транзакции. А обработчик события **OnBeginTransComplete** можно использовать для выполнения каких-то действий перед началом транзакции.

Метод **CommitTrans** завершает транзакцию и сохраняет ее результаты в базе данных. При успешном выполнении **CommitTrans** генерируется событие **OnCommitTransComplete** и свойство **InTransaction** устанавливается в **false**.

Метод **RollbackTrans** осуществляет откат: отменяет все изменения, сделанные на протяжении транзакции. При успешном выполнении **RollbackTrans** генерируется событие **OnRollbackTransComplete** и свойство **InTransaction** устанавливается в **true**.

Свойство **Attributes** описывает, как при соединении обрабатываются транзакции, оставшиеся незавершенными. Свойство является множеством, которое может быть пустым, или содержать одно или два значения: **xaCommitRetaining** и **xaAbortRetaining**.

Значение **xaCommitRetaining** приводит к тому, что при соединении незавершенные транзакции завершаются (срабатывает метод **CommitTrans**). Завершение транзакции автоматически вызывает начало новой транзакции.

Значение **xaAbortRetaining** приводит к тому, что при соединении незавершенные транзакции отменяются с откатом назад (срабатывает метод **Rollback-Trans**). Отмена транзакции автоматически вызывает начало новой транзакции.

Свойство **IsolationLevel** устанавливает уровень изоляции транзакции. Основные уровни:

ilReadUncommitted, ilBrowse	Видимы незафиксированные изменения, сделанные другими транзакциями
ilReadCommitted, ilCursorStability	Видимы только зафиксированные изменения, сделанные другими транзакциями
ilRepeatableRead	Изменения, сделанные другими транзакциями, не видны, но повторный запрос может выдать новые данные
ilSerializable, ilIsolated	Полная изоляция от других транзакций

10.4.4 Обзор компонентов наборов данных

Рассмотрим сначала особенности компонентов наборов данных ADO на примере **ADOTable**. Этот компонент может использоваться в приложениях вместо компонента **Table**, выполняющего аналогичные функции. Он вступает в контакт с указанной таблицей базы данных. База данных задается свойствами **ConnectionString** или **Connection**, как описано в разделе 10.4.2. Для управления таблицей в приложение вводится, помимо компонента **ADOTable**, обычный компонент источника данных **DataSource**, в свойстве **DataSet** которого задается имя компонента **in ADOTable**. Далее к этому источнику данных **DataSource** подключаются любые компоненты отображения данных.

Имя таблицы, как и в компоненте **Table**, задается свойством **TableName**. Однако, не все провайдеры поддерживают непосредственный доступ к таблице по ее имени. Они могут требовать доступ с помощью оператора SQL **SELECT**. Какой именно вариант доступа: прямой или через оператор **SELECT** будет использоваться, определяется свойством **TableDirect**. По умолчанию **TableDirect = false**, что означает автоматическое создание компонентом **ADOTable** соответствующего оператора **SELECT**.

Соединение с базой данных осуществляется так же, как и в компонентах BDE, методом **Open** или установкой в **true** свойства **Active**. Но при этом, если связь с базой данных осуществляется через компонент **ADOConnection**, надо учитывать описанную в разделе 10.4.3 взаимосвязь свойства **Active** компонента **ADOTable** и свойства **Connected** компонента **ADOConnection**.

В компоненте **ADOTable** имеется два свойства, характеризующих курсор, используемый при навигации по таблице. Одно из них — **CursorLocation** описано в разделе 10.4.3. Другое — **CursorType** описывает другие характеристики курсора. Это свойство может иметь значения:

ctUnspecified	Тип курсора не определен
ctOpenForwardOnly	Курсор может перемещаться по таблице только вперед. Этот тип курсора повышает производительность приложения

ctKeyset	При этом типе курсора записи, добавленные другими пользователями, невидимы, а записи, удаленные другими пользователями, недоступны. Этот тип используется по умолчанию
ctDynamic	Этот тип динамического курсора обеспечивает видимость всех изменений, сделанных другими пользователями: модификаций, удалений, вставок. Курсор может перемещаться по таблице вперед и назад
ctStatic	Этот тип статического курсора обеспечивает копирование записей. Изменения данных, сделанные другими пользователями, невидимы

Свойство **CacheSize** указывает, сколько записей заносится в локальный буфер оперативной памяти. По умолчанию **CacheSize** = 1. Если задать, например, **CacheSize** = 10, то при открытии базы данных в буфер загрузятся первые 10 записей. Пока будет идти работа с этими записями, все операции будут проводиться в оперативной памяти без обращения к базе данных. Если указатель таблицы вышел за пределы 10, то в память загрузятся следующие 10 записей и т.д. Естественно, что буферизация записей повышает эффективность работы.

Основные способы работы с **ADOTable** не отличаются от способов, работы с компонентом **Table**. Точно так же, как в компонентах **Table** и **Query**, двойной щелчок на компоненте вызывает редактор полей, в котором можно задать свойства отдельных полей, ничем не отличающиеся от полей компонентов BDE. Впрочем, одно печальное отличие есть: в компонентах ADO невозможно работать со словарями. Так что в каждом компоненте свойства полей приходится задавать вручную. Кроме того надо иметь в виду, что не все драйверы ADO могут работать с любыми типами полей. Например, драйвер Paradox ADO не работает с полями изображений. Так что в таблице Pers базы данных **dbP**, частично используемой в данной книге, не будет доступно поле **Photo** — фотографии сотрудников. Для драйвера **InterBase** (в наших примерах для базы данных **ib**) такого ограничения нет.

Из редактора полей так же, как в компонентах BDE, можно перетаскивать поля мышью на форму. При этом на форме автоматически будут создаваться соответствующие компоненты отображения данных.

Программный доступ к полям осуществляется так же, как в компонентах **Table** и **Query**: по индексу через свойство **Fields[i:integer]**, по имени поля с помощью метода **FieldByName("<имя>")**, по имени объекта поля.

Ограничения на вводимые значения в компоненте **ADOTable** можно обеспечивать только на уровне полей (см. раздел 9.5.3). Свойства, аналогичного **Constraints** в компонентах **Table** и **Query**, в **ADOTable** нет.

Упорядочивание отображаемых записей производится установкой свойства **IndexFieldNames**. В этом свойстве можно задавать любое сочетание имен полей, по которым вы хотите упорядочить отображение, разделяя их точками с запятой. Например, строка «Dep» упорядочит записи в таблице Pers по значению поля **Dep** — отдел. Строка «Dep;Fam;Nam;Pag» упорядочит записи по значению поля **Dep** — отдел, а внутри каждого отдела упорядочит по фамилии, имени и отчеству сотрудников. В отличие от свойства **IndexFieldNames** компонентов BDE, в компонентах ADO можно задавать любое сочетание полей, независимо от того, была ли индексирована таблица при ее создании по этим полям. В этом проявляется дополнительная гибкость компонентов ADO.

Фильтрация отображаемых данных может осуществляться так же, как в компонентах ADO, с помощью свойства **Filter**, в котором записываются условия отбора (см. раздел 9.5.5). Отличие от компонентов BDE заключается в том, что в компонентах ADO в строке **Filter** имена полей обязательно должны отделяться пробелом.

лами от операций отношения. Также пробелами должны окружаться логические операции **and** и **or**. Например, если в компонентах BDE фильтр может быть записан в виде:

```
(Year_b<=1960)and(Year_b>=1940)
```

то в компонентах ADO эта строка должна иметь вид:

```
(Year_b <= 1960) and (Year_b >= 1940)
```

Свойство **Filter** работает, если свойство **Filtered** = **true**.

Можно также использовать для фильтрации обработчик событий **OnFilterRecord** (см. раздел 9.5.5). Дополнительные возможности фильтрации обеспечивает свойство только времени выполнения **FilterGroup**. Этот параметр позволяет фильтровать записи, которые изменены, или должны были быть изменены, или удалены. Свойство может иметь следующие значения:

fgUnassigned	Не оказывает влияния на фильтрацию. Значение по умолчанию
fgNone	Отменяет текущую фильтрацию и делает видимыми все записи
fgPendingRecords	Отфильтровываются записи, которые были изменены, но еще не занесены в таблицу методом UpdateBatch или прерваны методом CancelBatch
fgAffectedRecords	Отфильтровываются последние измененные записи
fgFetchedRecords	Отфильтровываются записи, которые были изменены при последней очистке кэша
fgPredicate	Отфильтровываются только что удаленные записи
fgConflictingRecords	Отфильтровываются записи, которые должны были быть изменены, но это не получилось из-за ошибок

Методы, используемые при программировании работы с базой данных, в **ADO-Table** в основном те же, что в **Table** и **Query**. Навигация по таблице осуществляется методами **First**, **Next**, **Last** и **Prior**. При редактировании данных используются также методы, характерные для **Table**: **Insert**, **Edit**, **Post** и другие. Из методов поиска в ADO реализованы только методы **Locate** и **Lookup**. Их использование не отличается от описанного в разделе 9.11.6.

Из методов, отсутствующих в компонентах BDE, интересными представляют методы сохранения набора данных в файле и чтения его из файла. Сохранение в файле осуществляется методом **SaveToFile**:

```
void __fastcall SaveToFile(const WideString FileName,
                          TPersistFormat Format);
```

Параметр **FileName** указывает имя файла, в котором сохраняется набор данных. Параметр **Format** определяет формат файла. Этот параметр может принимать одно из двух значений: **pfADTG** — формат ADTG (Advanced Data Tablegram), или **pfXML** — формат XML (для версий ADO 2.1 и выше). Например:

```
ADOTable1->SaveToFile("Test.adt", pfADTG);
```

Чтение данных из файла осуществляется процедурой **LoadFromFile**:

```
void __fastcall LoadFromFile(const WideString FileName);
```

где **FileName** — имя файла. Загружать файл в набор данных можно даже при закрытом соединении с базой данных. В момент загрузки соединение автоматически откроется.

Методы **SaveToFile** и **LoadFromFile** удобно использовать для получения как бы мгновенного портрета данных на какой-то момент времени. Это может требоваться, например, для того, чтобы можно было восстановить запомненное, а затем из-за каких-то ошибок испорченное состояние базы данных.

Связь друг с другом компонентов **ADOTable**, работающих с разными таблицами, одна из которых главная, а другая — вспомогательная, осуществляется так же, как в компонентах **Table**, с помощью свойств **MasterSource** и **MasterFields** (см. раздел 9.10.1).

Теперь остановимся коротко на компоненте **ADOQuery** — аналоге компонента **Query**, используемого при работе с BDE. Этот компонент используется для выполнения произвольных запросов SQL. Его основное свойство **SQL**, содержащее запрос, и методы выполнения этого запроса ничем не отличаются от компонента **Query**. А соединение с базой данных, свойства и методы фильтрации и поиска аналогичны рассмотренным выше для компонента **ADOTable**.

Отличие от компонента **Query** заключается в методике работы с параметрами при динамических запросах. Если в запросе SQL указаны параметры, то в компоненте **Query** объекты типа **TParams**, соответствующие этим параметрам, расположены в подсвойстве **Items** свойства **Params**. Это массив параметров, причем доступ к значениям отдельных параметров во время выполнения может осуществляться или по индексу свойства **Items**, или по имени с помощью метода **ParamByName**, или еще несколькими способами (см. раздел 10.1.6.2). Например, возможны следующие операторы:

```
// значение параметра типа string
Query1->Params->Items[0]->Value = Edit1->Text;
Query1->Params->Items[0]->AsString = Edit1->Text;
Query1->ParamByName("Dep")->AsString = Edit1->Text;

// значение параметра типа integer
Query1->Params->Items[1]->Value = Edit2->Text;
Query1->Params->Items[1]->AsString = Edit2->Text;
Query1->ParamByName("PYear")->Value = Edit2->Text;
Query1->Params->FindParam("PYear")->Value = Edit2->Text;
Query1->Params->ParamValues["PYear"] = Edit2->Text;
```

В компоненте **ADOQuery** объекты, соответствующие параметрам, указанным в свойстве **SQL**, расположены в свойстве **Parameters** типа **TParameters**. Это тоже массив параметров, но его свойства и методы отличаются от свойств и методов **TParams**. Доступ к отдельным параметрам во время выполнения может осуществляться по индексу подсвойства **Items**, но при этом значение определяется только функцией **Value**, а методы **AsString**, **AsInteger** и т.п. отсутствуют. Доступ к отдельным параметрам может осуществляться по имени с помощью методов **ParamByName**, **FindParam**, **GetParamList**, **ParamValues** свойства **Parameters**, которые не отличаются от аналогичных методов свойства **Params** в **Query**. Например, возможны следующие операторы:

```
// значение параметра типа string
ADOQuery1->Parameters->Items[0]->Value = Edit1->Text;
ADOQuery1->Parameters->ParamByName("EDep")->Value = Edit1->Text;
ADOQuery1->Parameters->ParamValues["EDep"] = Edit1->Text;

// значение параметра типа integer
ADOQuery1->Parameters->Items[1]->Value = Edit2->Text;
ADOQuery1->Parameters->ParamByName("PYear")->Value = Edit2->Text;
ADOQuery1->Parameters->FindParam("PYear")->Value = Edit2->Text;
ADOQuery1->Parameters->ParamValues["PYear"] = Edit2->Text;
```

Еще одно различие параметров в компонентах **ADOQuery** и **Query** заключается в том, что во время проектирования компонент **Query** содержит только те пара-

метры, которые указаны в запросе SQL. А в **ADOQuery** предусмотрена возможность вводить параметры во время проектирования. Для этого надо нажать кнопку с многоточием около свойства **Parameters** в окне Инспектора Объектов, затем в появившемся окне редактора параметров щелкнуть правой кнопкой мыши и выбрать в контекстном меню раздел **Add** (вместо этого можно нажать соответствующую быструю кнопку).

Свойства параметров, которые вы можете устанавливать в окне Инспектора Объектов, если выделите какой-то из параметров в окне редактора параметров, несколько отличны от тех, которые свойственны параметрам компонента **Query**. Свойства **DataType**, **Name**, **Value** аналогичны свойствам **Query**. Свойство **Direction** в **ADOQuery** аналогично свойству **ParamType** в **Query**. Дополнительно для строковых параметров имеется свойство **Size** — число символов, а для числовых параметров имеются свойства **Precision** и **NumericScale** — общее максимальное число цифр и максимальное число цифр после десятичной запятой. Имеется также свойство **Attributes** — множество, которое может содержать значения **psSigned** — число может быть со знаком, **psNullable** — значение может быть нулевым, **psLong** — длинное двоичное значение.

Компонент **ADOStoredProc** является аналогом компонента **StoredProc**, используемого при работе с BDE. Этот компонент используется для выполнения хранящихся на сервере процедур. В целом он работает так же, как его аналог **StoredProc** (см. раздел 10.3.5.2).

Свойство, в котором задается имя выполняемой процедуры, называется **ProcedureName**, а не **StoredProcName**, как в **StoredProc**. После того, как вы зададите имя процедуры, в свойстве **Parameters**, аналогичном рассмотренному выше для компонента **ADOQuery**, появятся входные параметры процедуры. Выходные параметры представляются объектами полей компонента **ADOStoredProc**. Вы можете увидеть их и изменить их свойства, если сделаете двойной щелчок на компоненте **ADOStoredProc**, в появившемся окне Редактора Полей сделаете щелчок правой кнопкой мыши и выберете раздел **Add all fields**. В окне появятся поля, соответствующие всем выходным параметрам процедуры.

Таким образом, при работе с хранимыми процедурами вы сначала должны задать значения входных параметров, затем выполнить вызов процедуры оператором вида:

```
ADOStoredProc1->ExecProc();
```

а к возвращенным параметрам обращаться как к объектам полей компонента **ADOStoredProc**.

Мы рассмотрели, конечно, не все свойства и методы и не все компоненты ADO. Впрочем, надеюсь этого достаточно, чтобы ввести эти компоненты в любое из разработанных ранее приложений и на практике ознакомиться с их работой. Более подробные сведения и примеры вы можете найти в книгах [7] и [8].

10.5 Доступ к InterBase через InterBase Express (IBX)

10.5.1 Технология InterBase Express (IBX)

В библиотеке компонентов C++Builder 5 появилась страница **InterBase**, содержащая компоненты для работы с **InterBase** напрямую, минуя BDE. Эти компоненты обеспечивают повышенную производительность и позволяют использовать новые возможности сервера **InterBase**, недоступные обычным компонентам BDE.

Технология IBX обеспечивается следующими компонентами, расположенными на странице InterBase библиотеки:

IBTable	Компонент, используемый вместо Table для доступа к одной таблице набора данных
IBQuery	Компонент, аналогичный Query
IBStoredProc	Компонент для выполнения процедур, хранимых на сервере
IBDatabase	Обеспечивает соединение с базой данных InterBase
IBTransaction	Обеспечивает доступ ко всем богатым возможностям транзакций InterBase. Благодаря использования транзакций с опциями, наиболее подходящими к той или иной ситуации, повышается эффективность работы. Поддерживаются распределенные транзакции со множеством баз данных
IBUpdateSQL	Используется для изменений в таблицах только для чтения и для кэширования изменений. Позволяет проектировать нормализованные базы данных, не ограничивая возможностей приложения по изменению сложных наборов данных
IBSQL	Выполняет запросы SQL, минимизируя затраты буферизации и обмена данными с компонентами C++Builder. Обеспечивает наиболее эффективный доступ к данным InterBase
IBDataSet	Обеспечивает выполнение команды SELECT и выполняет команды SQL по вставке, удалению и изменению записей. Обеспечивает, как и IBSQL , эффективный доступ к данным
IBDatabaseInfo	Позволяет приложению затребовать информацию о базе данных и сервере InterBase, включая сведения о свойствах базы данных, производительности системы, о пользователях, соединенных с базой данных, и т.п.
IBSQLMonitor	Позволяет осуществлять отладку коммуникаций для ускорения работы проектов клиент/сервер и проектов с параллельными потоками
IBEvents	Обеспечивает асинхронную обработку событий сервера InterBase

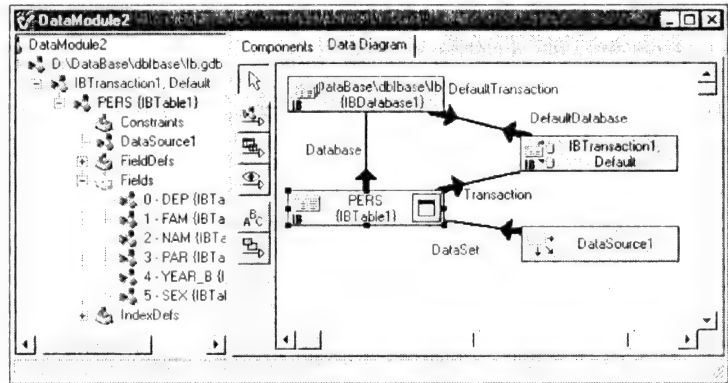
Подробное рассмотрение всех этих компонентов в рамках данной книги невозможно, так как требует детального знакомства с InterBase. Но некоторые основные моменты мы рассмотрим.

Для построения приложения на основе компонентов IBX на форме или в модуле данных должны быть прежде всего размещены компоненты **IBDatabase** и **IBTransaction**. Компоненты **IBDatabase** обеспечивают соединение с базой данных, а компоненты **IBTransaction** управляют транзакциями. К эти компонентам подключаются компоненты наборов данных: **IBTable**, **IBQuery**, **IBStoredProc**, **IBDataSet**, **IBSQL**, и др. Все эти компоненты подключаются заданием в них соответствующих значений свойства **Database** — имени компонента **IBDatabase**, и **Transaction** — имени компонента **IBTransaction**. А далее к компонентам наборов данных могут подключаться обычные компоненты источников данных **DataSource**, к которым в свою очередь подключаются обычные компоненты отображения и управления данными.

Эти связи поясняются приведенной на рис. 10.14 диаграммой модуля данных, построенного на компонентах IBX. Около связей указаны имена полей, которыми обеспечивается соответствующая связь.

Рис. 10.14.

Диаграмма модуля данных,
построенного на
компонентах IBX



10.5.2 Компоненты IBDatabase и IBTransaction

Соединение компонентов IBX с базой данных, как было сказано в разделе 10.5.1, осуществляется через компонент **IBDatabase**. Он выполняет для компонентов **ADOConnection** для компонентов ADO (см. раздел 10.4.3). База данных, с которой осуществляется соединение, задается свойством **DatabaseName**. Для локального набора данных InterBase задается имя файла базы данных. Например:

```
D:\DataBase\dbIbase\Ib.gdb
```

При нажатии кнопки с многоточием около этого свойства в окне Инспектора Объектов вызывается обычный диалог открытия файлов, который позволяет выбрать файл базы данных.

Для соединения с базой данных InterBase на удаленном сервере, использующем протокол TCP/IP, значение **DatabaseName** задается в виде:

```
<имя сервера>:<имя файла>
```

Для соединения с базой данных InterBase на удаленном сервере, использующем протокол NetBEUI, значение **DatabaseName** задается в виде:

```
\\<имя сервера>\<имя файла>
```

Для соединения с базой данных InterBase на удаленном сервере, использующем протокол SPX, значение **DatabaseName** задается в виде:

```
<имя сервера>@<имя файла>
```

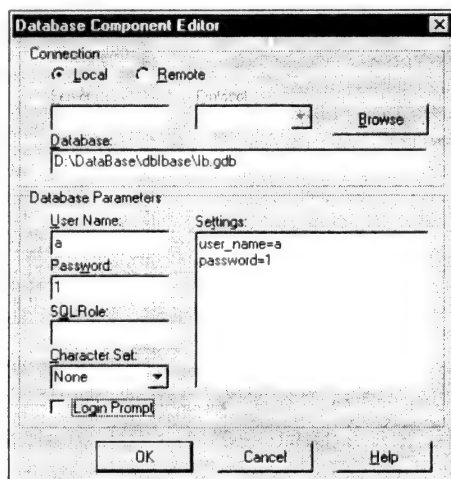
Возможен и другой способ задания базы данных — двойной щелчок на компоненте **IBDatabase**. В результате открывается окно, показанное на рис. 10.15. В нем вы можете не только задать в окошке Database имя базы данных (в этом может помочь кнопка поиска файла Browse), но и установить ряд свойств: в частности, имя пользователя (User Name) и пароль (Password). Результаты установок автоматически занесутся в окно Settings. Индикатор Login Prompt определяет появление или отсутствие окна запроса имени пользователя и пароля при каждом запуске приложения. Если вы укажете имя пользователя, пароль и снимете флажок индикатора Login Prompt, то запрос производиться не будет.

Свойство **SQLDialect** задает диалект SQL, используемый клиентом. Свойство **LoginPrompt** определяет появление диалога, запрашивающего имя пользователя и пароль при соединении с базой данных.

Свойство **Connected** открывает и закрывает соединение с базой данных. Соединение может управляться этим свойством или методами **Open** — открыть соединение, и **Close** — закрыть соединение. Эти методы переключают значение **Connected** соответственно в **true** и в **false**.

Рис. 10.15.

Окно задания базы данных и ее свойств



Существует также метод **CloseDataSets**, закрывающий соединение всех компонентов наборов данных, соединенных с данным компонентом **IBDatabase**, но не закрывающий соединение самого **IBDatabase**.

Свойство **Connected** компонента **IBDatabase** связано со свойствами **Active** компонентов наборов данных, подключенных к данному **IBDatabase**. Это односторонняя связь, аналогичная описанной в разделе 10.4.3 для компонента **ADOConnection**.

Теперь рассмотрим компонент **IBTransaction**, который позволяет организовывать транзакции. Выполнение транзакции начинается методом **StartTransaction**. Перед вызовом этого метода полезно проверить значение свойства **InTransaction**. Если оно равно **true**, это значит, что не закончена предыдущая транзакция. В этом случае вызов новой транзакции без завершения предыдущей методами **Commit** или **Rollback** приведет к генерации исключения.

После вызова **StartTransaction** сервер хранит все изменения, вставки, удаления записей, пока не будет выполнен метод **Commit** или метод **Rollback**. Метод **Commit** завершает транзакцию, запоминая все сделанные на протяжении ее изменения в базе данных. Метод **Rollback** производит откат, т.е. отменяет все изменения данных, произведенные на протяжении текущей транзакции, и завершает эту транзакцию.

Таким образом, организация транзакции осуществляется по следующей схеме:

```
IBDatabase1->Open();
DataModule2->IBTransaction1->StartTransaction();
операторы изменения данных, удаления и вставки записей
...
if(<проверка правильности изменений, запрос пользователю на фиксацию
изменений и т.п.>)
    IBTransaction1->Commit();
else IBTransaction1->Rollback();
```

Например:

```
IBTransaction1->StartTransaction();
IBDatabase1->Open();
...
if (Application->MessageBox(
    "Действительно хотите изменить сохранить изменения?",
    "Подтвердите сохранение изменений",
```

```

        MB_YESNOCANCEL+MB_ICONQUESTION) == IDYES)
    IBTransaction1->Commit();
else IBTransaction1->Rollback();

```

Завершение транзакции методами **Commit** и **Rollback** сбрасывает свойство **Active** компонента **IBTransaction** в **false**. Между этим свойством **IBTransaction** и аналогичными свойствами подключенных к **IBTransaction** компонентов такое же взаимодействие, которое описано выше для **IBDatabase**. Поэтому для последующей активизации подключенных компонентов надо принимать те же меры, которые были описаны для **IBDatabase**.

Теперь рассмотрим некоторые свойства компонента. **DefaultDatabase** — задает компонент **IBDatabase**, используемый по умолчанию при выполнении транзакции.

Свойство **IdleTimer** задает отрезок времени в секундах, через который, если транзакция не завершена, совершается действие по умолчанию. Это действие задается свойством **DefaultAction**, которое может принимать значения:

taRollback	Откат транзакции
taCommit	Фиксация результатов транзакции в базе данных
taRollbackRetaining	Откат транзакции с сохранением ее контекста. Доступно только начиная с InterBase 6
taCommitRetaining	Фиксация результатов транзакции в базе данных и сохранение контекста текущей транзакции

В момент окончания отведенного отрезка времени генерируется событие **OnIdleTimer**, обработчик которого можно использовать для каких-то дополнительных действий, например, для запроса пользователя о дальнейших действиях.

Свойства только для чтения **DatabaseCount** и **Databases** задают соответственно число компонентов **IBDatabase**, вовлеченных в текущую транзакцию, и индексированный список этих компонентов.

10.5.3 Компоненты наборов данных **IBTable**, **IBQuery**, **IBStoredProc**

Все компоненты наборов данных подключаются к компонентам **IBDatabase** и **IBTransaction** через свои свойства **Database** и **Transaction**, как было указано в разделе 10.5.1.

Компонент **IBTable** является эквивалентом обычного компонента **Table**, используемого при работе с BDE, и у него практически нет свойств, которых бы не было в **Table**. Пожалуй, имеет смысл отметить только свойство **UniDirectional**, указывающее, возможно ли применение для данной таблицы двунаправленного курсора. Надо также сказать, что, к сожалению, в компоненте **IBTable** невозможно использовать словари полей. Так что свойства полей приходится устанавливать вручную, пользуясь Редактором Полей, вызываемом двойным щелчком на компоненте.

Методы компонента **IBTable** тоже в основном совпадают с **Table**. Однако, из методов поиска реализованы только методы **Locate** и **Lookup**. Их использование не отличается от описанного в разделе 9.11.6.

Компонент **IBQuery** является аналогом обычного компонента **Query**, используемого при работе с BDE. Он позволяет общаться с базой данных на языке SQL. Свойства и методы **IBQuery** подобны свойствам и методам **Query**.

Компонент **IBStoredProc** является аналогом компонента **StoredProc**, используемого при работе с BDE. Этот компонент используется для выполнения храни-

мых на сервере процедур. В целом он работает так же, как его аналог **StoredProc** (см. раздел 10.3.5.2).

Имя выполняемой процедуры задается свойством **StoredProcName**, как и в **StoredProc**. После того, как вы зададите имя процедуры, в свойстве **Params** появятся выходные и входные параметры процедуры. Отличие от аналогичного свойства компонента **StoredProc** заключается в последовательности размещения параметров. В **StoredProc** сначала располагаются входные параметры, а затем выходные. А в **IBStoredProc** сначала расположены выходные параметры, а затем входные. Впрочем, это не имеет значения, если для доступа к параметрам используется метод **ParamByName**.

Таким образом, практически в любом приложении можно просто заменить компоненты **StoredProc** на **IBStoredProc**, добавив, конечно, в приложение компоненты **IBDatabase** и **IBTransaction**.

Глава 11

Обработка и документирование данных

11.1 Многомерный анализ данных — компоненты Decision Cube

11.1.1 Настройка компонентов приложения

Для анализа многофакторной информации, получаемой из базы данных, в C++Builder имеются специальные компоненты, размещенные на странице библиотеки Decision Cube. При анализе данные представляются в виде так называемого многомерного куба (метакуба) или куба решений. Каждое измерение этого куба соответствует одному полю. Например, данные по сотрудникам организации, содержащиеся в таблице *Pers*, могут анализироваться по подразделениям, по признаку пола, по году рождения, по тому, работают ли они в управлении или в производственном отделе. Таким образом возникает четырехмерный куб. При анализе может возникнуть желание узнать, сколько человек работают в управлении и в производстве, сколько всего в организации мужчин и женщин, как распределяются мужчины и женщины в управлении и на производстве, распределение сотрудников по годам рождения и т.п.

Таких вопросов может быть великое множество и в приложении невозможно предусмотреть какие-то меню, кнопки и иные управляющие элементы, которые бы охватывали все, что может захотеться знать пользователю, принимающему решения на основе сведений, почерпнутых из базы данных. Желательно иметь инструмент, с помощью которого пользователь мог бы сам наглядно формулировать любые запросы, получая на них ответы в удобной табличной или графической форме. Именно таким инструментом и является система Decision Cube.

Эта система включает следующие компоненты:

DecisionCube	Реализует многомерный куб данных
DecisionGraph	Отображает графики, соответствующие выбору, сделанному пользователем в многомерном кубе
DecisionGrid	Отображает в табличном виде данные, соответствующие выбору, сделанному пользователем в многомерном кубе
DecisionPivot	Дает возможность пользователю закрывать и открывать отдельные измерения куба
DecisionQuery	Определяет набор данных, используемый для построения куба. Аналог компонента Query , приспособленный для задач Decision Cube
DecisionSource	Источник данных, аналогичный DataSource , но приспособленный для задач Decision Cube

Рассмотрим применение всех этих компонентов на примере работы с базой данных *ib*, использовавшейся в главе 10.

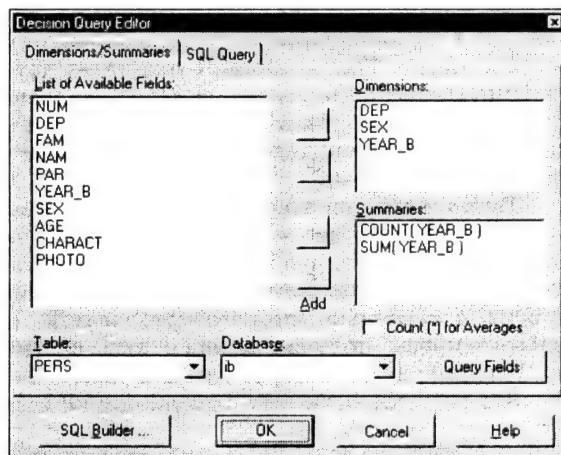
Начнем с простого приложения, работающего с таблицей Pers. Начните новое приложение и перенесите на форму компоненты **DecisionQuery**, **DecisionCube**, **DecisionSource** и **DecisionGrid**. Компоненты **DecisionQuery**, **DecisionCube** и **DecisionSource** — невизуальные. Так что разместить их можно в любом месте формы. А компонент отображения данных **DecisionGrid** — визуальный. Задайте в его свойстве **Align** значение **alClient**, чтобы он заполнил всю поверхность формы.

Свяжите размещенные компоненты друг с другом цепочкой ссылок, похожей на ту, которая связывает обычные компоненты, работающие с базами данных. В компонент отображения данных **DecisionGrid1** надо в его свойстве **DecisionSource** ввести ссылку на компонент источника данных **DecisionSource1**. В источник данных **DecisionSource1** надо в его свойстве **DecisionCube** ввести ссылку на метакуб **DecisionCube1**. А в метакубе **DecisionCube1** надо в свойстве **DataSet** сослаться на набор данных **DecisionQuery1**. Таким образом создается цепочка: набор данных (**DecisionQuery**) \Rightarrow метакуб (**DecisionCube**) \Rightarrow источник данных (**DecisionSource1**) \Rightarrow отображение данных (**DecisionGrid**). От привычной цепочки, используемой при работе с базами данных, эта цепь отличается только одним звеном — наличием метакуба между набором и источником данных.

Начать проектирование следует с настройки набора данных **DecisionQuery1**. Сделайте на этом компоненте двойной щелчок. Перед вами откроется окно редактора запроса Decision Cube, показанное на рис. 11.1.

Рис. 11.1.

Окно редактора запроса Decision Cube



В окошке Database вы должны выбрать из выпадающего списка базу данных, с которой собираетесь работать, а в окошке Table — таблицу этой базы данных. После этого в списке всех доступных полей (List Of Available Fields) вы увидите список полей таблицы. Отберите из них те, которые вы хотите анализировать. Для отбора надо выделить соответствующее поле в списке List Of Available Fields и нажать кнопку со стрелкой вправо между этим списком и списком Dimensions — измерения. Тем самым вы создаете измерение метакуба, соответствующее выбранному полю.

Последовательность занесения полей в список Dimensions не безразлична. Дело в том, что первое измерение будет в дальнейшем соответствовать столбцам таблицы, а остальные — строкам. Причем доступ пользователя к строкам будет тем проще, чем выше расположено поле в списке Dimensions. Так что в нашем примере желательно сформировать список в той последовательности, которая указана на рис. 11.1.

В список Summaries следует добавить те суммарные характеристики, которые будут отображаться в ячейках таблицы или нужны для расчета характеристик, отображаемых в ячейках. Пусть мы хотим отображать в таблице число сотрудни-

ков по отделам, полу, возрасту, и, кроме того, хотим иметь возможность отображать средний год рождения. При расчете среднего года рождения автоматически будет рассчитываться сумма годов рождения и количество содержащихся в каждом разделе годов рождения, т.е. число сотрудников. Так что нам достаточно обеспечить расчет среднего года рождения **Year_b**, а число сотрудников рассчитается автоматически. Выделите в списке поле **Year_b** и нажмите кнопку со стрелкой вправо около окна **Summaries**. Перед вами возникнет выпадающее меню с тремя разделами: **sum** — сумма, **count** — количество и **average** — среднее. Выберите **average**. Однако в окно **Summaries** при этом занесется не **AVG(YEAR_B)**, а **SUM(YEAR_B)** и **COUNT(YEAR_B)**. Это занеслись промежуточные величины, необходимые для подсчета среднего значения.

На странице **SQL Query** (рис. 11.2) вы можете увидеть запрос **SQL**, который сформировался в результате ваших действий. В нашем случае он будет иметь вид:

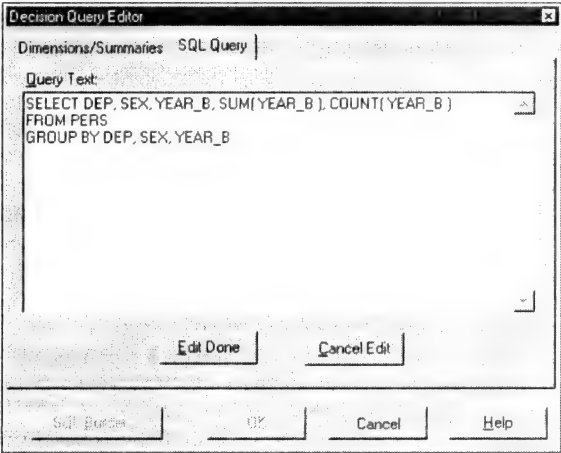
```
SELECT DEP, SEX, YEAR_B, SUM( YEAR_B ), COUNT( YEAR_B )
FROM PERS
GROUP BY DEP, SEX, YEAR_B
```

В этом запросе после ключевого слова **SELECT** указаны поля измерений (**Dep**, **Sex**, **Year_b**), затем указаны суммарные характеристики. В конце оператора указана группировка — **GROUP BY**. Тут обязательно должны быть повторены все поля, соответствующие измерениям, и в той же последовательности, в которой они фигурировали в списке **SELECT**. Вы можете при желании отредактировать этот запрос, нажав предварительно кнопку **Edit Query**, или просто начав вводить какие-то символы. При этом надпись на кнопке **Edit Query** изменится на **Edit Done**. После завершения редактирования нажмите на эту кнопку, и введенные вами изменения зафиксируются, или вам будет выдано сообщение об ошибке, если вы что-то написали неправильно. Кнопка **Edit Cancel** позволяет прервать редактирование.

Первая из суммарных характеристик, указанных в запросе **SQL**, появится в ячейках таблицы при первом предъявлении ее пользователю. В приведенном выше запросе **SQL** первой указана сумма годов рождения — вспомогательная величина, абсолютно бессмысленная для пользователя. Поэтому желательно отредактировать запрос, переставив величину **COUNT(YEAR_B)** перед **SUM(YEAR_B)**. Если вы после этого вернетесь на страницу **Dimension/Summaries**, то увидите, что последовательность этих величин в окне **Summaries** изменилась (см. рис. 11.1).

Внизу окна на рис. 11.1 и 11.2 вы видите кнопку **SQL Builder**. Она позволяет вызвать визуальный построитель запросов **SQL**, в котором вы можете сформировать запрос. Его результаты автоматически отобразятся на страницах окна редактора запроса **Decision Cube**.

Рис. 11.2.
Страница **SQL Query** окна редактора запроса **Decision Cube**



Завершив всю работу в окне редактора запроса, нажмите ОК. Вы вернетесь в проектируемое вами приложение, а имя базы данных и запрос SQL автоматически занесутся в свойства **Database** и **SQL** компонента **DecisionQuery**.

Итак, основной компонент — **DecisionQuery** мы настроили. Теперь можете его активизировать. Измените значение его свойства **Active** на **true**. При этом прямо в процессе проектирования в компоненте **DecisionGrid1** отобразятся данные (рис. 11.3).

Рис. 11.3.

Форма простого приложения

Sex	Деп	Цех 1	Цех 2	Sum
ж	1	1		2
м	2	3	4	9
Sum	3	4	4	11

Позднее мы разберемся, как работать с этими данными. А пока давайте настроим форму их отображения. Прежде всего хотелось бы иметь в заголовках русские надписи. Сделайте двойной щелчок на компоненте **DecisionCube1**, или нажмите кнопку с многоточием в окне Инспектора Объектов около свойства **DimensionMap** этого компонента, или щелкните на компоненте правой кнопкой мыши и выберите в контекстном меню раздел **Decision Cube Editor**. Во всех этих вариантах вы попадете в окно, представленное на рис. 11.4.

На странице **Decision Settings** этого окна вы можете, выбирая каждое поле в списке **Available Fields**, задать для него в окошке **Display Name** имя, которое будет фигурировать в заголовках компонентов отображения данных. Окошко **Type** указывает тип выделенного в списке **Available Fields** элемента: является ли он измерением куба или суммарной характеристикой. Значение в этом окошке изменять невозможно. Выпадающий список **Active Type** определяет, когда загружается соответствующая информация: **As Needed** — когда требуется ее отображать, **Active** — всегда, **Inactive** — никогда. В большинстве случаев следует выбирать значение **As Needed**.

Окошко **Format** позволяет задать строку форматирования отображения. Выпадающий список **Grouping** дает возможность выбрать отображение всех значений (**None**), или только лежащих в определенных границах: **Year** — год, **Quarter** — квартал, **Month** — месяц, **Single Value** — одномерное отображение. Все это относится к данным, распределенным во времени, и в нашем примере использоваться не может.

Рис. 11.4.

Страница **Decision Settings** редактора куба решений

Decision Cube Editor

Dimension Settings | Memory Control

Available Fields

- DEP*
- SEX*
- YEAR_B*
- COUNTALL*
- SUM*
- Average of PERS YEAR B*

Display Name: средний г.р.

Type: Measurement

Active Type: As Needed

Format:

Grouping: None

OK Cancel Help

Страница Memory Control окна редактора куба решений особого интереса не представляет. Если вы откроете ее, то имеет смысл обратить внимание только на опцию Designer Data Options, которая указывает, что именно должно отображаться в процессе проектирования. На отображение данных в процессе выполнения приложения эта опция не влияет.

Редактор куба решений позволяет вам задать описанным выше способом настройку (в частности, Display Name — отображаемые русские названия) сразу для всех компонентов отображения данных, если их несколько в вашем приложении. Но каждый компонент отображения данных позволяет произвести дополнительную индивидуальную настройку отображения. Обратите внимание в компоненте **DecisionGrid** на свойство **Dimensions**. Нажав кнопку с многоточием около него вы попадете в редактор списка измерений, подобный тем, с которыми вы уже не раз встречались на протяжении чтения этой книги. В нем вы можете для каждого измерения задать отображаемое имя (**DisplayName**), формат и в свойстве **Subtotal** указать, надо ли для данного измерения отображать промежуточные суммарные данные: например, число сотрудников по каждому отделу, по полу, по каждому году рождения. Последнее, видимо, не надо, так что для поля года рождения следует задать **Subtotal = false**. А для среднего года рождения полезно задать формат «##0.#». Он означает, что отображение будет проводиться до первого знака после запятой и что, если какой-то категории сотрудников нет и, следовательно, их средний год рождения 0, то этот 0 будет отображаться в ячейке таблицы.

Из свойств **DecisionGrid** обратите также внимание на **DefaultColWidth** — ширина колонки таблицы по умолчанию. Это свойство задает ширину по умолчанию. В процессе выполнения приложения пользователь сможет по своему усмотрению изменять ширину колонок.

Свойства **LabelColor** и **LabelSumColor** позволяют задавать цвета соответственно основных ячеек таблицы и строк и столбцов сумм. Последние, наверное, неплохо выделить цветом, отличным от основных ячеек.

11.1.2 Управление выполняющимся приложением

Теперь настройка приложения закончена и вы можете выполнить его (рис. 11.5). Вы видите, что в каждом измерении имеется индикатор с символом «+» — развернуть, или «-» — свернуть. Строки можно полностью свернуть (рис. 11.5.а), и тогда отображается только число сотрудников по отделам. Впрочем, свертывание можно продолжить, свернув и первое измерение — отделы. В этом случае таблица отобразит всего одну ячейку — число сотрудников в учреждении. А можно развернуть второе измерение (рис. 11.5 б) и информация станет двумерной, классифицируя число сотрудников по отделам и по полу. Если ввести еще одно измерение (рис. 11.5 б), то информация сортируется по отделам, по полу и по годам рождения.

Пользователь может в процессе выполнения приложения управлять отображением промежуточных сумм по отдельным измерениям и всех суммарных данных. Если вы щелкнете правой кнопкой мыши на пустом заголовке таблицы или на пустом месте таблицы, то всплывет меню, состоящее из одного раздела — **Subtotal on/off**. Это переключатель, позволяющий включать и выключать отображение всех суммарных данных. Иное контекстное меню всплывет при щелчке правой кнопкой мыши на одном из заголовков измерений. Оно включает два раздела: раздел **Display Data and SubTotal** — отображение и данных, и промежуточных сумм по данному измерению, и раздел **Display Data Only** — уничтожение промежуточных сумм данного измерения и отображению только данных.

Щелкнув правой кнопкой мыши на заголовке какого-то подраздела, вы увидите меню всего с одним разделом — **Drill in to this value**. Выбор этого раздела равноценен удалению из таблицы соответствующего измерения.

Рис. 11.5.

Приложение во время выполнения с различной степенью детализации информации

а) Многомерный анализ данных

Отдел				
Бухгалтерия	Цех 1	Цех 2	Sum	
3	4	4	11	

б) Многомерный анализ данных

Пол	Отдел			
ж	Бухгалтерия	Цех 1	Цех 2	Sum
ж		1	1	2
м		2	3	4
Sum		3	4	11

в) Многомерный анализ данных

Пол	г.р.	Отдел			
ж	1955	Бухгалтерия	Цех 1	Цех 2	Sum
ж	1961		1	1	1
ж	Sum		1	1	2
м	1930		1	1	2
м	1937		1	1	1
м	1950		1	1	1
м	1955		1	1	1
м	1960		1	1	1
м	1962		1	1	1
м	1975		1	1	2
м	Sum		2	3	9
Sum			3	4	11

Таковы возможности пользователя, если в процессе проектирования вы включили в компоненте **DecisionGrid** в свойстве **Options** опцию **cgPivotable**. Попробуйте включить эту опцию. Вы увидите, что возможности пользователя многократно увеличились. Щелкните теперь в выполняющемся приложении правой кнопкой мыши на пустом месте заголовка или на пустом месте таблицы. Перед вами всплывет меню, показанное на рис. 11.6 а. В нем вы можете отметить те измерения, которые хотите наблюдать, и таблица перестроится соответствующим образом. Отметив раздел среднего года рождения, вы перейдете от отображения числа сотрудников к отображению среднего года рождения.

Рис. 11.6.

Контекстное меню приложения

а) Subtotals on/off

<input checked="" type="checkbox"/> Отдел
<input checked="" type="checkbox"/> Пол
<input type="checkbox"/> г.р.
<input checked="" type="checkbox"/> число чел.
<input type="checkbox"/> Sum
<input type="checkbox"/> средний г.р.

б) Отдел

<input checked="" type="checkbox"/> Display Data and Subtotals
<input type="checkbox"/> Display Data Only
<input checked="" type="checkbox"/> Отдел
<input checked="" type="checkbox"/> Пол
<input type="checkbox"/> г.р.
<input checked="" type="checkbox"/> число чел.
<input type="checkbox"/> Sum
<input type="checkbox"/> средний г.р.

Верхний раздел меню на рис. 11.6 позволяет включать и отключать отображение сумм, как промежуточных, так и полных.

Если вы щелкнете правой кнопкой мыши на заголовке какого-то измерения, перед вами всплывет меню, показанное на рис. 11.6 б. Раздел **Display Data and Sub-Total** обеспечивает отображение и данных, и промежуточных сумм по данному измерению, а раздел **Display Data Only** приводит к уничтожению промежуточных сумм данного измерения и отображению только данных. Таким образом, с помощью меню рис. 11.6 вы можете убирать или отображать как промежуточные суммы по отдельным измерениям, так и все суммы.

Пользователь может не только манипулировать видимостью отдельных измерений. Он может перетащить мышью заголовок какого-то раздела и поменять таким образом последовательность измерений. Он может даже перетаскивать измерения из строк в столбцы и обратно, полностью меняя структуру таблицы. Мне представляется, что подобная свобода в руках не очень опытного пользователя может полностью его запутать. Впрочем, все эти возможности можно отменить, включив в компоненте **DecisionGrid** опцию **cgPivotable** в свойстве **Options**.

Выше было рассказано об управлении приложением с помощью всплывающих меню. Но возможности пользователя можно еще расширить, если установить в **true** свойство **ShowCubeEditor** компонента **DecisionGrid**. В этом случае пользователю при щелчке правой кнопкой мыши вместо списка видимых измерений (см. рис. 11.6) будет предлагаться вызов редактора измерений, т.е. диалогового окна, показанного ранее на рис. 11.4. Там он может менять тексты заголовков, форматы представления данных и т.п. Правда, мне нелегко представить квалификацию пользователя, которому это могло бы быть полезно.

В целом, думается, что приложение, использующее описанные выше возможности **Decision Cube**, должно быть снабжено хорошей встроенной справочной системой, которая помогала бы пользователю.

11.1.3 Компонент **DecisionPivot**

Компонент **DecisionPivot** обеспечивает значительно более удобное управление измерениями, чем описано выше. Добавьте на форму вашего приложения этот компонент, установите его свойство **Align** в **alTop** и укажите в свойстве **DecisionSource** источник данных **DecisionSource1**. И это все! Правда, поскольку теперь управлять измерениями будет компонент **DecisionPivot**, то, вероятно, имеет смысл выключить опцию **cgOutliner** в свойстве **Options** компонента **DecisionGrid**. При этом из таблицы исчезнут кнопки со знаками «+» и «-», которые вы можете видеть на рис. 11.5 и которые ранее позволяли пользователю сворачивать и разворачивать измерения. В том же свойстве **Options** имеет смысл выключить также опцию **cgPivotable**, о которой говорилось в предыдущем разделе.

Запустите приложение и посмотрите, как оно теперь работает (рис. 11.7).

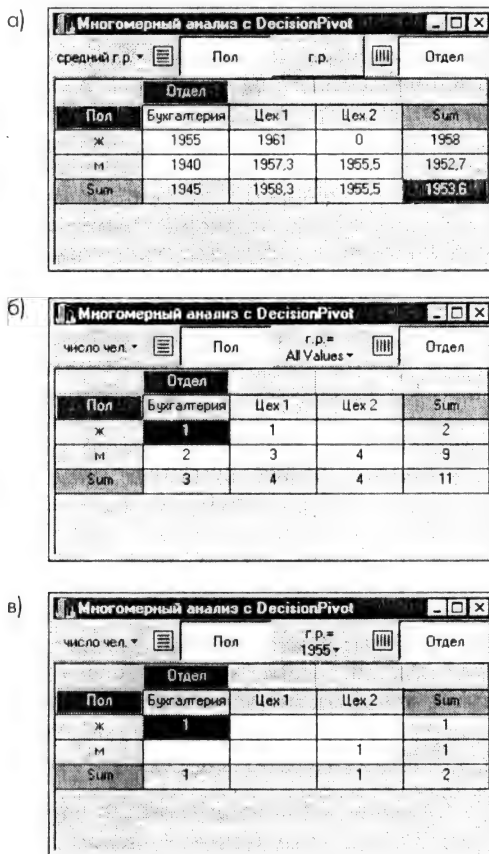
Панель компонента **DecisionPivot** имеет три группы кнопок. Слева расположена кнопка, показывающая величину, отображаемую в ячейках таблицы. Эта кнопка снабжена выпадающим списком, позволяющим менять отображаемую величину (в нашем примере — число сотрудников или средний год рождения). Следующая группа кнопок соответствует измерениям, которые размещены в строках таблицы (в нашем примере Пол и г.р.). Правая группа кнопок соответствует измерениям, размещенным в столбцах таблицы (Отдел). Нажатие той или иной кнопки включает или выключает показ соответствующего измерения.

Кнопки можно перетаскивать мышью, меняя последовательность измерений или перемещая измерения из строк в столбцы и обратно. Такое перемещение из строк в столбца или обратно можно также осуществлять, щелкнув правой кнопкой мыши на кнопке панели и выбрав из контекстного меню соответствующий раздел.

Компонент **DecisionPivot** предоставляет интересную возможность фиксации значения измерения. Если вы щелкнете правой кнопкой мыши на какой-то кнопке панели и выберете из всплывшего меню раздел **Drilled in**, то надпись на кнопке

Рис. 11.7.

Приложение с компонентом DecisionPivot



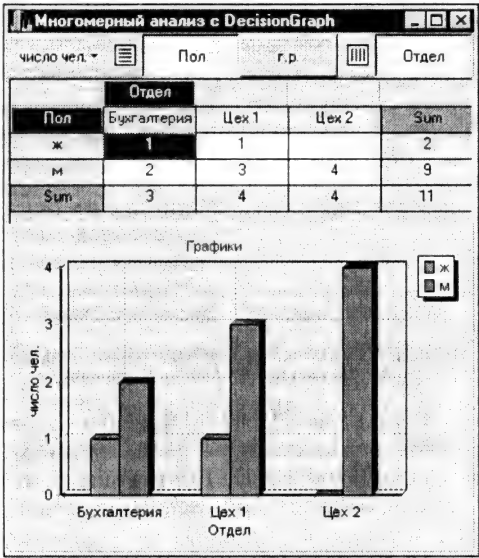
изменится (см. рис. 11.7 б). По данному измерению будут отражаться суммарные данные. Но если после этого вы нажмете на эту кнопку, то появится меню, содержащее разделы **Open Dimention** — открыть измерение (т.е. вернуться к обычной форме отображения), **All Values** — отображать суммарные данные, а также в этом меню будет список всех значений данного измерения. Вы можете выбрать какое-то значение (рис. 11.7 в) и отобразятся только данные, относящиеся к этому значению. Конечно, в нашем примере это не очень нужно. Но если бы мы работали, например, с базой данных каких-то товаров или услуг, то это позволило бы нам избирательно получить информацию по какому-то конкретному товару или услуге.

11.1.4 Компонент DecisionGraph

В заключение краткого рассмотрения компонентов Decision Cube остановимся на компоненте **DecisionGraph**. Этот компонент позволяет вводить в приложение диаграммы и графики. Перенесите в свое приложение компонент **DecisionGraph** (рис. 11.8) и установите его свойство **DecisionSource** равным **DecisionSource1**. Компонент можно разместить в нижней части таблицы **DecisionGrid** и установить в нем свойство **Align** равным **alBottom**. Запустите свое приложение. Вы увидите, что теперь данные отображаются не только в таблице, но и графически. Причем переменные, откладываемые по осям, и вид диаграмм автоматически изменяются при переключении пользователем измерений.

Основные свойства компонента **DecisionGraph** во многом аналогичны свойствам компонента **Chart** (см. раздел 3.4.6 главы 3). При настройке вы увидите, что

Рис. 11.8.
Приложение с диаграммами



серии заносятся в компонент автоматически, так что вам остается только установить элементы оформления.

В заключение рассмотрения многомерного анализа данных с помощью Decision Cube надо отметить, что в рассмотренных примерах мы не написали ни одного оператора. Конечно, это возможно только в чисто демонстрационном приложении. Управлять компонентами системы Decision Cube можно программно, задавая во время выполнения значения свойств компонентов, используя их методы и обработчики событий. Особых сложностей в этом нет, но подробное рассмотрение приемов программирования работы с компонентами Decision Cube выходит за рамки данной книги (увы, объем ее надо ограничивать). Так что посмотрите немногие методы и события компонентов Decision Cube во встроенной справке C++Builder.

11.2 Создание отчетов

Для создания отчетов в C++Builder включена система QuickReport. Компоненты этой системы размещены на странице QReport палитры компонентов.

QuickReport использует генератор отчетов, состоящих из множества полос. Полоса (band) — это область отчета или раздел, содержащий некоторый текст, изображения, графики, диаграммы и т.п. Полоса является контейнером для других компонентов, вносящих в отчет информацию или графику.

Если полоса и размещенные на ней компоненты связаны с базой данных, то содержание этой полосы печатается столько раз, сколько соответствующих записей имеется в источнике данных. Таким образом, достаточно расположить компоненты, связанные с данными, на полосе, а печатаемые значения и их количество будут автоматически управляться базой данных.

Как в компонентах, связанных с данными, вы можете задавать головную и вспомогательную таблицы, так и между полосами можно задавать аналогичные связи. Таким образом, все возможности, реализуемые в приложениях, реализуются с тем же успехом и в отчетах.

Давайте зададимся задачей построить отчет по уже многократно использованной нами в главе 9 базе данных dbP, первая страница которого показана на рис. 11.9. Отчет, озаглавленный «КАДРОВЫЙ СОСТАВ ПРЕДПРИЯТИЯ ПО СОСТОЯНИЮ НА ...» должен включать в себя следующие разделы:

Заголовок раздела	Информация, распечатываемая в разделе
ОТДЕЛЫ	Список всех отделов
СПИСОЧНЫЙ СОСТАВ ОТДЕЛОВ	Состоит из ряда подразделов, каждый из которых имеет свой подзаголовок «СОТРУДНИКИ ОТДЕЛА ...», после которого следует список фамилий сотрудников
ЛИЧНЫЕ ДЕЛА	Состоит из ряда подразделов, каждый из которых имеет свой подзаголовок «СОТРУДНИКИ ОТДЕЛА...», после которого следует информация о фамилии, имени, отчестве сотрудника, его пол, год рождения, фотография сотрудника и его характеристика
ВЫВОДЫ	Содержит произвольный текст, вводимый пользователем

В разделе «ЛИЧНЫЕ ДЕЛА» каждый подраздел «СОТРУДНИКИ ОТДЕЛА...» должен начинаться с новой страницы. С новой страницы должен также начинаться раздел «Выводы». Все страницы должны иметь нижний колонтитул, в котором слева печатается надпись «Кадровый состав», а справа — номер страницы.

Рис. 11.9.

Первая страница
разрабатываемого отчета

**КАДРОВЫЙ СОСТАВ ПРЕДПРИЯТИЯ
ПО СОСТОЯНИЮ НА 20.04.00**

ОТДЕЛ

Руководитель
 Цех 1
 Цех 2

СПИСОЧНЫЙ СОСТАВ ОТДЕЛОВ

СОТРУДНИКИ ОТДЕЛА Руководитель
 Беловая Беловая Беловая
 Иваша Иваша Иваша
 Ивашин Ивашин Ивашин

СОТРУДНИКИ ОТДЕЛА Цех 1
 Борисов Борисов Борисов
 Иваша Иваша Иваша
 Пивов Пивов Пивов
 Петров Петр Петров
 Огород Огород Огород

СОТРУДНИКИ ОТДЕЛА Цех 2
 Бедров Бедров Бедров
 Иваша Иваша Иваша
 Харитонов Харитонов Харитонов


ЛИЧНЫЕ ДЕЛА

СОТРУДНИКИ ОТДЕЛА Руководитель
 Беловая Беловая Беловая
 1966 г.р. пол >

Характеристика
 Беловая Беловая Беловая,
 1961 г.р.,
 сотрудник фотографии

Б.Б. Беловая совсем не работает, не выполняет
 организационные и производственные функции.

Ивашин О.К. И.И. Иваша



Кадровый состав

Строя этот отчет, мы по ходу дела рассмотрим необходимые нам свойства различных компонентов.

Основным компонентом, на котором строится весь отчет, является **QuickRep**. Он предоставляет ряд возможностей по управлению создаваемым отчетом, включая формирование заголовка, полос, шрифтов, установок принтера и др. Этот компонент является визуальным и после его соединения с базой данных может использоваться как контейнер полос, составляющих отчет.

Компонент **QuickRep** имеет ряд свойств, определяющих характеристики печати отчета:

PrinterSetting	Задает число копий отчета и диапазон печатаемых страниц
Page	Задает размер страницы PaperSize (можно установить заказной размер — Custom и определить длину и ширину страницы свойствами Length и Width), ее ориентацию и поля
Options	Определяет, надо ли печатать верхний колонтитул первой страницы (FirstPageHeader) и нижний колонтитул последней (LastPageFooter)
Units	Задает единицу измерения размеров страницы, полей и т.п.: миллиметры, дюймы, пиксели и т.д.
Zoom	Масштаб печати в процентах
ReportTitle	Заголовок окна предварительного просмотра

Свойство **DataSet** определяет набор данных, к которому подключается отчет. Этим набором может являться компонент типа **TTable**, **TQuery** и т.п.

Компонент **QuickRep** имеет два основных метода: **Preview** — предварительный просмотр, и **Print** — печать. Предварительный просмотр и даже печать отчета можно осуществлять и в процессе проектирования. Для этого надо щелкнуть правой кнопкой мыши на компоненте **QuickRep** и из всплывшего меню выбрать команду **Preview**. Перед вами откроется окно предварительного просмотра, в котором, в частности, имеется кнопка печати.

Компоненты **QRLabel**, **QRMemo**, **QRRichText**, **QRShape**, **QRImage**, размещаемые на полосах отчета, являются аналогами обычных компонентов — **Label**, **Memo**, **RichEdit**, **Shape**, **Image**. Основной особенностью соответствующих компонентов **QuickReport** является их способность печататься в тех полосах отчета, в которых они размещены. Компоненты имеют два свойства, отсутствующих в обычных компонентах: **Frame** и **Size**.

Свойство **Frame** имеет ряд подсвойств, определяющих рамку вокруг компонента: **Color** — цвет, **Style** — стиль, **Width** — ширина, **DrawBottom**, **DrawLeft**, **DrawRight**, **DrawTop** — определяют наличие рамки соответственно внизу, слева, справа и сверху компонента.

Свойство **Size** имеет подсвойства, определяющие размер и место размещения компонента при печати. Все определяется в единицах измерения, заданных свойством **Units** компонента **QuickRep**.

Некоторые компоненты имеют свойство **AlignToBand** — выравнивание в полосе. Если это свойство установить в **true**, то компонент будет выровнен по краю полосы, заданному свойством **Alignment**: **taLeftJustify** — влево, **taCenter** — по центру, **taRightJustify** — вправо.

Давайте начнем построение нашего отчета. Перенесите на форму компонент **QuickRep**. Поместите на форму компонент **Table**, назовите его **TDep** и свяжите с таблицей **Dep** базы данных **dbP**. Перенесите на форму компонент **DataSource**, назовите его **DSDep** и свяжите с **TDep**. Перенесите на форму еще один компонент

Table, назовите его **TPers** и свяжите с таблицей **Pers** базы данных **dbP**. Установите между двумя таблицами обычную связь, чтобы **TDep** была головной таблицей, а **TPers** связывалась с ней по полю **Dep** (задайте в **TPers** соответствующие значения свойств **IndexName**, **MasterSource**, **MasterFields** — см. раздел 9.10.1). Установите свойства **Active** компонентов **Table** в **true**.

Установите в свойстве **DataSet** компонента **QuickRep** имя компонента **TDep**, связав его тем самым с головной таблицей. Теперь рассмотрим одно из основных свойств компонента **QuickRep** — **Bands**. Оно имеет ряд подсвойств:

HasTitle	Имеется полоса заголовка отчета, которая печатается один раз в начале отчета
HasDetail	Имеется полоса детализации, которая печатается столько раз, сколько записей в нее передается
HasPageHeader	Имеется верхний колонтитул (заголовок) на каждой странице отчета
HasPageFooter	Имеется нижний колонтитул на каждой странице отчета
HasColumnHeader	Имеется заголовок печатаемой таблицы

Для построения нашего отчета надо установить в **true** подсвойства **HasTitle** и **HasPageFooter** — полосы заголовка и нижнего колонтитула. На компоненте **QuickRep** появятся слабо видимые полосы с соответствующими надписями. На них и будут размещаться компоненты, которые отображают ту или иную информацию.

Разместите на полосе заголовка метку **QRLabel** и в ее свойстве **Caption** запишите первую часть заголовка отчета «КАДРОВЫЙ СОСТАВ ПРЕДПРИЯТИЯ». Выровняйте эту метку по центру полосы, установив ее свойство **AlignToBand** в **true**, а свойство **Alignment** в **taCenter**. Выберите увеличенный размер шрифта заголовка (например, 14). Установите жирный шрифт.

Ниже поместите компонент **QRSysData**. Этот компонент позволяет отображать в отчете системные данные. Его основное свойство **Data**, которое может принимать следующие значения:

qrsDate	текущая дата
qrsDateTime	текущие дата и время
qrsDetailCount	число записей в базе данных
qrsDetailNo	текущий номер записи в базе данных
qrsPageNumber	номер текущей страницы
qrsReportTitle	заголовок отчета
qrsTime	текущее время

Свойство **Text** компонента **QRSysData** определяет текст, предшествующий отображаемой величине.

В нашем случае свойство **Data** установите равным **qrsDate**, а свойство **Text** задайте равным «ПО СОСТОЯНИЮ НА ». Это обеспечит внесение в заголовок текущей даты. Выровняйте **QRSysData** по центру так же, как делали это для метки **QRLabel**, и так же установите шрифт.

На полосе нижнего колонтитула разместите метку **QRLabel**, выровненную влево, с надписью «Кадровый состав», которая должна появляться в колонтитуле.

Установите в ее свойстве **Frame** подсвойство **DrawTop** в **true**. Это обеспечит появление линии, отделяющей колонтитул от текста.

Разместите на той же полосе нижнего колонтитула компонент **QRSysData**, задав его свойство **Data** равным **qrsPageNumber** и выровняв вправо. Этот компонент будет отображать номер страницы.

Можете щелкнуть правой кнопкой мыши на компоненте **QuickRep** и, выбрав из всплывшего меню команду **Preview**, полюбоваться достигнутым результатом. Теперь займемся собственно текстом отчета. Для этого надо разместить на компоненте **QuickRep** дополнительные полосы. Мы будем это делать, перенося на него соответствующие компоненты полос со страницы **QReport** палитры компонентов.

Для первого раздела нашего отчета — «ОТДЕЛЫ» нам надо организовать печать заголовка раздела и далее циклическую печать названий отделов. Это может сделать полоса детализации в виде компонента **QRSubDetail**. Перенесите ее на **QuickRep** и давайте рассмотрим некоторые ее свойства.

Свойство **DataSet** определяет набор данных, к которому должна подключаться полоса. Поскольку в данном случае она должна обеспечивать просмотр всех записей таблицы **Dep**, в свойстве **DataSet** надо указать таблицу **TDep**. Этого достаточно, чтобы обеспечить циклическую печать полосы.

Свойство полосы **Bands** имеет два подсвойства: **HasFooter** определяет полосу, которая будет напечатана после окончания циклов, и **HasHeader** определяет полосу заголовка, которая будет напечатана перед началом циклической печати. Нам требуется установить в **true** только **HasHeader**. В появившейся полосе **Group Header** поместите метку **QRLabel** с заголовком раздела «Отделы» и соответствующими установками шрифта, а в самой полосе детализации поместите компонент **QRDBText** — метку, связанную с данными. В ней, как и в других компонентах, связанных с данными, надо задать набор данных **DataSet** и его поле **DataField**, которое должно отображаться. В данном случае **DataSet** = **TDep** и **DataField** = **Dep**.

Можете опять осуществить предварительный просмотр отчета и убедиться, что первый раздел отчета печатается правильно.

Во втором разделе — «СПИСОЧНЫЙ СОСТАВ ОТДЕЛОВ» нам надо организовать два вложенных цикла печати: внешний по отделам и внутренний по сотрудникам очередного отдела. Для внешнего цикла повторяете все операции, которые делали для предыдущего раздела: переносите в отчет компонент **QRSubDetail**, связываете его с таблицей **TDep**, устанавливаете в **true** подсвойство **HasHeader** свойства **Bands**. На полосе заголовка раздела помещаете метку **QRLabel** с заголовком «СПИСОЧНЫЙ СОСТАВ ОТДЕЛОВ». На полосе детализации размещаете метку **QRLabel** с текстом «СОТРУДНИКИ ОТДЕЛА» и рядом с ней — метку **QRDBText**, настроив ее на поле **Dep** таблицы **TDep**.

Теперь нам надо организовать вложенный цикл, чтобы для каждого отдела пробегать по относящимся к нему записям таблицы **Pers**. Для задания таких вложенных циклов в компоненте **QRSubDetail** предусмотрено свойство **Master**. Оно определяет головную полосу для данной полосы. Это аналогично тому, как задается головная таблица для вспомогательной. Свойство **Master** позволяет организовывать внутренние циклы печати для каждого напечатанного значения головной полосы. Свойство **PrintBefore** определяет в этом случае, печатаются ли значения внутреннего цикла до (при **PrintBefore** = **true**) или после печати очередного значения внешнего цикла.

Чтобы все это реализовать, поместите в отчет еще один компонент **QRSubDetail** и свяжите его с таблицей **TPers**. Его свойство **Master** установите в **QRSubDetail2** — имя второй полосы **QRSubDetail**, являющейся головной во втором разделе. Свойство **PrintBefore** установите в **false**. На эту новую полосу поместите метки **QRDBText**, настроив их на поля **Fam**, **Nam** и **Par** таблицы **TPers**.

Запустите предварительный просмотр отчета и убедитесь, что все работает как надо.

Теперь третий раздел отчета «ЛИЧНЫЕ ДЕЛА» не представит для вас сложности. В нем точно так же надо организовать внешний цикл печати по отделам и внутренний — по сотрудникам отдела. В этом втором цикле для отображения фотографий надо использовать компонент **QRDBImage**, а для печати характеристик — компонент **QRDBRichText**. Оба эти компонента должны быть настроены на соответствующие поля таблицы **TPers**. Осталось выполнить одно требование задания — каждый подраздел раздела «ЛИЧНЫЕ ДЕЛА», относящийся к новому подразделению, должен начинаться с новой страницы. Это можно сделать, воспользовавшись свойством полосы детализации **ForceNewPage** — форсировать переход на новую страницу. В полосе внешнего цикла установите это свойство в **true**. Тогда по окончании каждого внешнего цикла будет осуществляться переход на новую страницу.

Последний раздел вашего отчета — «ВЫВОДЫ». Перенесите в отчет новую полосу **QRSubDetail**, разместите метку **QRLabel** с текстом «ВЫВОДЫ» и поместите компонент **QRRichText**, в котором пользователь сможет отображать написанный им текст.

Форма, которую вы сделали — вспомогательная. Пользователь не будет ее видеть. А теперь надо сделать главную форму приложения, с которой будет работать пользователь. В ней должно быть меню (или кнопки), которое позволит осуществлять предварительный просмотр и печать отчета. Кроме того на этой форме должно быть окно редактирования, в котором пользователь может написать выводы. Этот текст и должен отображаться в отчете на последней странице.

Если вы посмотрите в окне Инспектора Объектов свойства компонента **QRRichText**, то увидите свойство **ParentRichEdit**. В этом свойстве можно указать обычный компонент **RichEdit**, текст которого автоматически будет переноситься в текст компонента **ParentRichEdit**. Именно этот родительский компонент **RichEdit** и надо разместить на главной форме.

Давайте займемся проектированием главной формы. Прежде всего убедитесь, что форма вашего отчета невидима (свойство **Visible** равно **false**). Назовите эту форму **FRep**. Сохраните модуль вашего отчета, задав его имя, например, **URep**. Это имя, как и имя формы, потребуется вам в дальнейшем для ссылок.

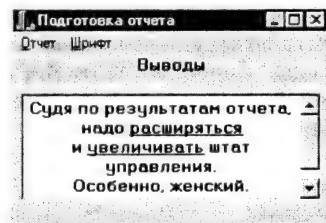
Добавьте в приложение новую форму (команда **File | New Form**). Назовите форму **FMain**. Сохраните ее модуль, дав ему, например, имя **UMain**. Эти имена нам потребуются в дальнейшем для ссылок.

Выполните команду **Project | Options**, в открывшемся окне Опций проекта перейдите на страницу **Forms** и в списке **Main Form** выберите главной текущую форму **FMain**. Форма **FRep** должна быть вспомогательной.

Перенесите на новую форму компоненты главное меню **MainMenu**, **FontDialog**, **RichEdit** и метку. В меню введите раздел **Отчет** с подразделами **Просмотр** и **Печать**, и раздел **Шрифт**. Расположите все примерно так, как показано на рис. 11.10, на котором форма изображена во время выполнения приложения.

Рис. 11.10.

Главная форма приложения генерации отчетов



Теперь осталось связать друг с другом две формы и написать небольшие команды управления.

Для связи модуля формы отчета **URep** с модулем **UMain** включите в модуль **URep** соответствующую директиву препроцессора **#include**. Это можно сделать непосредственно, или с помощью вызова из **URep** команды главного меню **File | Include Unit Hdr**.

В компоненте **QRRichText** формы **FRep** раскройте выпадающий список в свойстве **ParentRichEdit**. В этом списке должна появиться ссылка на компонент **RichEdit** формы **FMain**. Установите это свойство, чтобы связать окна редактирования друг с другом.

Перейдите в модуль главной формы и той же командой **File | Include Unit Hdr** свяжите его с модулем **URep**.

В обработчик команды меню **Просмотр** вставьте оператор

```
FRep->QuickRep1->Preview();
```

В обработчик команды меню **Печать** вставьте оператор

```
FRep->QuickRep1->Print();
```

В обработчик команды меню **Шрифт** вставьте операторы

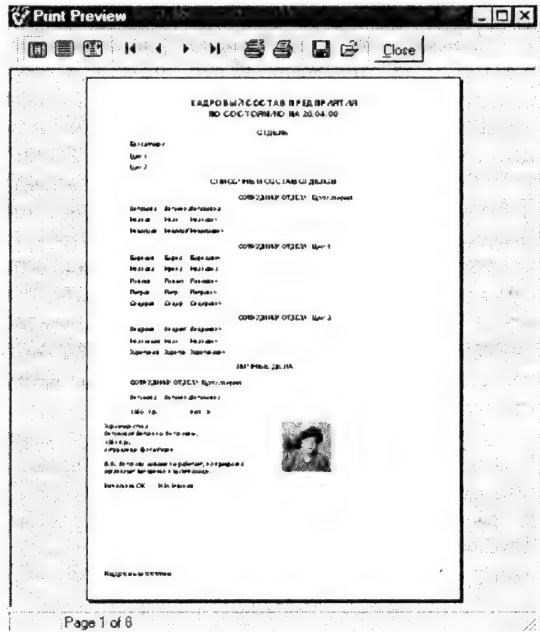
```
if(FontDialog1->Execute())
    RichEdit1->SelAttributes->Assign(FontDialog1->Font);
RichEdit1->SetFocus();
```

Эти операторы обеспечат некоторые минимальные возможности форматирования в окне **RichEdit1**. Конечно, в настоящей программе этого мало и надо бы действительно использовать богатые возможности компонента **RichEdit** (см. раздел 3.2.4).

На этом разработка приложения для подготовки отчета закончена. Можете запускать свое приложение в эксплуатацию. С его помощью в любой момент можно распечатать полный отчет о текущем состоянии базы данных. При выборе раздела меню **Просмотр** пользователю будет предъявляться окно предварительного просмотра, показанное на рис. 11.11. В этом окне с помощью быстрых кнопок, расположенных вверху на инструментальной панели, пользователь сможет изменять масштаб отображения страницы (три левые кнопки), перемещаться по страницам

Рис. 11.11.

Окно предварительного просмотра отчета



(кнопки навигатора), осуществлять установку принтера и печать, сохранять отчет или открывать файл какого-то из прошлых отчетов. Кнопка Close закрывает окно предварительного просмотра, после чего можно в окне редактирования главной формы написать текст в раздел «Выводы» и напечатать отчет.

11.3 Использование серверов COM для документирования данных

При составлении отчетов, содержащих сведения, черпаемые из баз данных, можно использовать компоненты системы QuickReport, описанные в разделе 11.2. Однако, QuickReport накладывает на форму отчетов достаточно жесткие ограничения. На основе QuickReport можно разработать приложения для некоторых стандартных отчетов с часто обновляемыми данными. Но нередко хотелось бы иметь значительно большую свободу при компоновке и написании отчетов, хотелось бы иметь возможность вставлять данные в некий произвольный текст и в произвольной форме. Такую свободу дает программа Windows Word, широко используемая каждым, имеющим дело с персональными компьютерами. Поэтому представляет интерес рассмотреть методику совместного использования Word и приложений C++Builder при обработке и документировании информации, содержащейся в базах данных. В данном разделе мы проиллюстрируем некоторые способы управления программой Word из приложений C++Builder.

Взаимодействие с Word, Excel и многими другими распространенными программами, входящими в стандартную установку Word и Microsoft Office, может осуществляться из приложений C++Builder 5 с помощью компонентов, размещенных в библиотеке на странице Servers. Эти компоненты, отображающие множество импортируемых серверов COM, рассмотрены в разделе 6.4.4 главы 6. А в данном разделе мы рассмотрим демонстрационный пример приложения, использующий сервер Word при работе с базами данных.

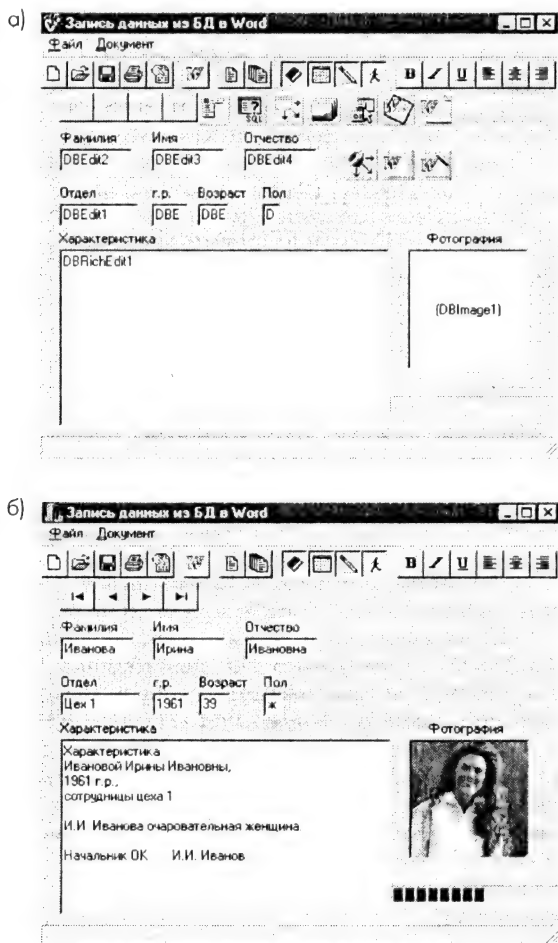
Пример такого приложения приведен на рис. 11.12. Попробуйте сделать аналогичное и испытать на нем описанные методы работа с серверами COM. Соответствующий пример приведен на диске, приложенном к книге.

Приложение позволяет просматривать таблицу Pers базы данных **ib** и заносить в активный документ Word, начиная с текущей позиции курсора, сведения о сотруднике из выбранной пользователем записи или заносить информацию о всех записях таблицы. При этом пользователь может отбирать, какая именно информация будет заноситься в документ. Фамилия, имя и отчество сотрудника заносятся в любом случае. Наименование подразделения, в котором работает сотрудник, заносится в виде «Сотрудник подразделения ...» или «Сотрудница подразделения ...» только в случае, если пользователем нажата соответствующая кнопка на инструментальной панели (ее имя в приведенном далее коде — **TBDep**). Занесение в документ года рождения, характеристики и фотографии также определяется тем, нажаты или не нажаты соответственно кнопки **TBYear**, **TBCharact**, **TBPhoto** на инструментальной панели. На рис. 11.12 б все эти кнопки изображены в нажатом состоянии в середине панели. Фрагмент документа Word, подготовленного приложением, приведен на рис. 11.13.

Внесенный в документ текст может быть автоматически отформатирован. Стиль шрифта при форматировании определяется нажатием кнопок инструментальной панели **TBBold** — жирный, **TBItalic** — курсив, **TBUnderline** — подчеркнутый. Выравнивание введенного текста определяется тем, какая из кнопок **TBLeft** (влево), **TBCenter** (по центру) или **TBRight** (вправо) нажата. На рис. 11.12 б эти кнопки крайние правые на инструментальной панели.

Рис. 11.12.

Форма демонстрационного приложения (а)
и окно приложения во время выполнения (б)



Приложение может управлять сервером Word, выполняя такие стандартные действия, как создание нового документа, открытие файла, сохранение документа в файле, предварительный просмотр, печать.

Работа с Word может протекать невидимо для пользователя (кроме, конечно, стандартных диалогов открытия файла и т.п.). Но пользователь может, нажав соответствующую кнопку на инструментальной панели, сделать окно Word видимым, перейти в него, отредактировать текст, написать дополнительный текст, т.е. может нормальным образом работать с Word.

Теперь перейдем к описанию построения приложения. Оно включает в себя следующие компоненты. Связь с базой данных **ib** осуществляет компонент **Query1**, в свойстве **SQL** которого записан оператор

```
Select * from Pers Order by Fam, Nam, Par
```

Источником данных, связанным с **Query1**, является **DataSource1**. С ним связаны компоненты отображения данных **DBEdit1** — **DBEdit7**, отображающие текстовые поля записи, компонент **DBRichEdit1**, отображающий характеристику, и компонент **DBImage1**, отображающий фотографию.

На форме расположен компонент **ActionList1**, в котором описаны основные действия, выполняемые в приложении:

Действие	Описание	Обработчик
AAI	Внесение в документ всех записей	TForm1->AAIExecute
AExit	Выход	TForm1->AExitExecute
ANew	Создание нового документа	TForm1->ANewExecute
AOpen	Открытие файла	TForm1->AOpenExecute
APreview	Предварительный просмотр документа	TForm1->APreviewExecute
APrint	Печать документа	TForm1->APrintExecute
ARecord	Внесение в документ одной записи	TForm1->ARecordExecute
ASave	Сохранение документа	TForm1->ASaveExecute
AWord	Сделать Word видимым	TForm1->AWordExecute

Компонент **ActionList1** связан с компонентом типа **TImageList**, содержащим пиктограммы для быстрых кнопок и разделов меню.

В верхней части формы расположена инструментальная панель — компонент **ToolBar1** с множеством быстрых кнопок. Имена некоторых из них, которые используются в коде приложения, были приведены выше при описании функционирования приложения. Меню, как всегда, создано с помощью компонента **MainMenu**.

Внизу формы расположена полоса состояния, в которой отображаются подсказки кнопок инструментальной панели и разделов меню. Для отображения подсказок в приложение введен компонент **ApplicationEvents1**, обработчик события **OnHint** которого обеспечивает отображение свойств **Hint** кнопок и разделов меню в полосе состояния.

Рис. 11.13.

Фрагмент документа, подготовленного демонстрационным приложением

Иванов Иван Иванович

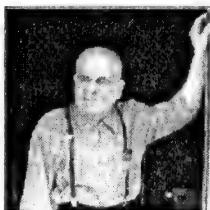
1950 года рождения

Сотрудник подразделения 'Бухгалтерия'

Характеристика
Иванова Ивана Ивановича,
1950 г.р.,
сотрудника бухгалтерии

И.И. Иванов отличный работник.

Начальник ОК И.И. Иванов



Диаграмма, отображающая ход формирования документа при занесении в него всех записей, реализована компонентом **ProgressBar1** (см. раздел 3.4.3). Его свойство **Visible** установлено в **false**, чтобы компонент не был виден. Его изображение появляется на форме только во время формирования документа.

На форме размещены серверы COM: компоненты **WordApplication1**, **WordDocument1**, **WordFont1** и **WordParagraphFormat1**. Их свойства **AutoConnect** установлены в **false**. Свойство **ConnectKind** в компоненте **WordApplication1** установлено равным **ckRunningOrNew**, а в остальных серверах равно **ckAttachToInterface**.

Ниже приведен текст кода приложения с некоторыми не имеющими значения купюрами.

Заголовочный файл:

```
class TForm1 : public TForm
{
    __published:      // IDE-managed Components
    ...
private: // User declarations
    void __fastcall DocumentSearch(void);
public:    // User declarations
    __fastcall TForm1(TComponent* Owner);
};
...
```

Файл реализации:

```
#include <Clipbrd.hpp>

void __fastcall TForm1::DocumentSearch(void)
{
    //Проверка наличия открытого документа
    if(WordApplication1->Documents->Count == 0)
    {
        Application->MessageBox("В Word нет открытого документа",
                                "Команда не выполнена",
                                MB_OK + MB_ICONEXCLAMATION);
        Abort();
    }
    WordDocument1->ConnectTo(WordApplication1->ActiveDocument);
}
//-----
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Query1->Open();
    /* Выключение проверок синтаксиса и грамматики,
       чтобы не замедлять работу Winword*/
    WordApplication1->Options->CheckSpellingAsYouType = false;
    WordApplication1->Options->CheckGrammarAsYouType = false;
    if(WordApplication1->Documents->Count == 0)
    {
        ASave->Enabled = false;
        APreview->Enabled = false;
        APrint->Enabled = false;
        ARecord->Enabled = false;
        AAll->Enabled = false;
    }
}
//-----
void __fastcall TForm1::ANewExecute(TObject *Sender)
{
    //Открытие нового документа
    WordApplication1->Documents->Add(EmptyParam, EmptyParam);
    ASave->Enabled = true;
}
```

```

    APreview->Enabled = true;
    APrint->Enabled = true;
    ARecord->Enabled = true;
    AAll->Enabled = true;
}
//-----
void __fastcall TForm1::AAllExecute(TObject *Sender)
{
    //Перенос в документ всех записей
    TBookmark SavePlace;
    //Закладка на текущей записи
    SavePlace = Query1->GetBookmark();
    Query1->First();
    //Сообщение в строке состояния
    StatusBar1->SimpleText = "Идет формирование документа";
    //Настройка диаграммы
    ProgressBar1->Max = Query1->RecordCount;
    ProgressBar1->Position = 0;
    ProgressBar1->Visible = true;
    //Цикл по записям
    while (! Query1->Eof)
    {
        ARecordExecute(Sender);
        ProgressBar1->Position = ProgressBar1->Position + 1;
        Query1->Next();
    }
    //Возвращение на текущую запись
    Query1->GotoBookmark(SavePlace);
    //Очистка закладки, полосы состояния и диаграммы
    Query1->FreeBookmark(SavePlace);
    StatusBar1->SimpleText = "";
    ProgressBar1->Visible = false;
}
//-----
void __fastcall TForm1::ARecordExecute(TObject *Sender)
{
    TVariant snw = "\n";
    //Перенос в документ одной записи
    DocumentSearch();
    WordApplication1->Selection->InsertAfter(snw);
    WordApplication1->Selection->InsertAfter(
        TVariant(Query1FAM->AsString + ' ' +
        Query1NAM->AsString + ' ' +
        Query1PAR->AsString + '\n'));
    WordApplication1->Selection->InsertAfter(snw);
    if (TBYear->Down)
    {
        WordApplication1->Selection->InsertAfter(
            TVariant(Query1YEAR_B->AsString + " года рождения \n"));
        WordApplication1->Selection->InsertAfter(snw);
    }
    if (TBDep->Down)
    {
        if (Query1SEX->AsString == 'М')
            WordApplication1->Selection->InsertAfter(
                TVariant("Сотрудник подразделения '" +
                Query1DEP->AsString + "'\n"));
        else WordApplication1->Selection->InsertAfter(
            TVariant("Сотрудница подразделения '" +
            Query1DEP->AsString + "'\n"));
    }
    if (TBCharact->Down)
    {

```

```

WordApplication1->Selection->InsertAfter(snew);
WordApplication1->Selection->InsertAfter(
    TVariant(DBRichEdit1->Text));
WordApplication1->Selection->InsertAfter(snew);
}
//Форматирование шрифта введенного текста
WordFont1->ConnectTo(WordApplication1->Selection->Font);
if(TBUnderline->Down)
    WordFont1->Underline = wdUnderlineSingle;
else WordFont1->Underline = wdUnderlineNone;
if(TBBold->Down)
    WordFont1->Bold = 1;
else WordFont1->Bold = 0;
if(TBItalic->Down)
    WordFont1->Italic = 1;
else WordFont1->Italic = 0;
//Форматирование абзацев введенного текста
WordParagraphFormat1->ConnectTo(
    WordApplication1->Selection->ParagraphFormat);
if(TBLeft->Down)
    WordParagraphFormat1->Alignment = wdAlignParagraphLeft;
if(TBCenter->Down)
    WordParagraphFormat1->Alignment = wdAlignParagraphCenter;
if(TBRight->Down)
    WordParagraphFormat1->Alignment = wdAlignParagraphRight;
TVariant Direction = wdCollapseEnd;
WordApplication1->Selection->Collapse(&Direction);
//Перенос в документ фотографии
if(TBPhoto->Down)
{
    Clipboard()->Assign(DBImage1->Picture);
    WordApplication1->Selection->Paste();
    WordApplication1->Selection->InsertAfter(snew);
}
}
//-----
void __fastcall TForm1::ASaveExecute(TObject *Sender)
{
    DocumentSearch();
    WordApplication1->Dialogs->Item(wdDialogFileSaveAs)->
        Show(EmptyParam);
}
//-----
void __fastcall TForm1::AWordExecute(TObject *Sender)
{
    //Открытие и соединение с Word,
    //если пользователь случайно закрыл его
    WordApplication1->Connect();
    //Включение видимости сервера
    WordApplication1->Visible = true;
}
//-----
void __fastcall TForm1::ApplicationEvents1Hint(TObject *Sender)
{
    //Отображение подсказок в строке состояния
    StatusBar1->SimpleText = Application->Hint;
}
//-----
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
    /*Разрыв соединения с базой данных при завершении приложения*/
    Query1->Close();
}

```

```

//-----
void __fastcall TForm1::APrintExecute(TObject *Sender)
{
    //Печать
    DocumentSearch();
    WordApplication1->Visible = true;
    WordApplication1->Dialogs->Item(wdDialogFilePrint)->
        Show(EmptyParam);
}
//-----
void __fastcall TForm1::APreviewExecute(TObject *Sender)
{
    //Предварительный просмотр документа
    DocumentSearch();
    WordDocument1->PrintPreview();
    WordApplication1->Visible = true;
}
//-----
void __fastcall TForm1::AExitExecute(TObject *Sender)
{
    //Выход
    Close();
}
//-----
void __fastcall TForm1::AOpenExecute(TObject *Sender)
{
    //Открытие файла
    WordApplication1->Visible = true;
    if(WordApplication1->Dialogs->Item(wdDialogFileOpen)->
        Show(EmptyParam) == -1)
    {
        ASave->Enabled = false;
        APreview->Enabled = false;
        APrint->Enabled = false;
        ARecord->Enabled = false;
        AAll->Enabled = false;
    }
}

```

Приведенный код содержит подробные комментарии. К тому же, многие фрагменты этого кода уже были подробно разобраны при обсуждении свойств и методов серверов в разделе 6.4.4. Так что ограничимся только краткими дополнительными пояснениями.

Поскольку в тексте приложения в процедуре **TForm1->ARecordExecute** используется объект буфера обмена **Clipboard**, через который осуществляется пересылка в документ фотографии, то в оператор включена директива компилятора, подключающая модуль **Clipbrd**. Без ссылки на этот модуль компилятор не понял бы идентификатора **Clipboard**.

Основные процедуры, связанные с серверами, прокомментированы в тексте и вряд ли нуждаются в дополнительных пояснениях. Имеет смысл только обратить внимание на то, что процедура **TForm1->AAllExecute**, соответствующая переносу в документ информации о всех записях таблицы, только организует цикл по записям, но сама перенос информации в документ не делает, обращаясь для этого к процедуре **TForm1->ARecordExecute**, которая переносит в документ текущую запись. Чтобы цикл не сбивал текущую запись, установленную ранее пользователем, перед началом цикла с помощью метода **GetBookmark** создается закладка **Save-Place** типа **TBookmark**, соответствующая текущей записи. После завершения цикла методом **GotoBookmark** происходит возврат на эту закладку, после чего она уничтожается методом **FreeBookmark**.

Перед началом цикла настраивается и делается видимой диаграмма **ProgressBar1**. Ее свойство **Position** изменяется на каждом цикле. После завершения цикла **ProgressBar1** опять делается невидимой.

Процедура **DocumentSearch**, которая объявлена в заголовочном модуле как функция-элемент класса, обеспечивает проверку наличия в Word хотя бы одного открытого документа. Это необходимо делать при выполнении таких операций, как сохранение файла, его печать и т.п., которые бессмысленны в отсутствие документа. Поэтому в начале подобных процедур вызывается **DocumentSearch**.

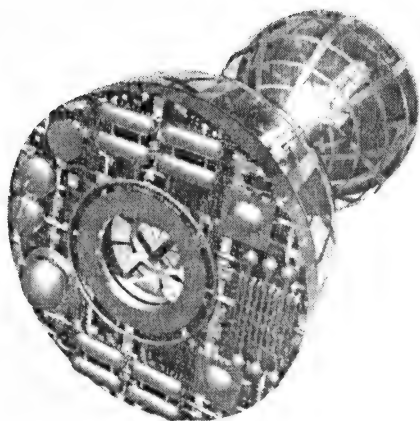
На этом мы закончим рассмотрение данного примера. Сделайте сами подобный пример и поэкспериментируйте с ним. Учтите, что компиляция и вообще все операции с приложением, использующим серверы, могут выполняться заметно медленнее обычного. Но это только во время отладки. Если вы выполните сделанное приложение не из среды C++Builder, то никаких задержек выполнения не будет.

Конечно, данный пример чисто демонстрационный, но его можно было бы легко развить. Можно добавить в него рассмотренные в предыдущих главах способы фильтрации данных, быстрого поиска нужной информации и т.п. Вы можете использовать все возможности, предоставляемые C++Builder для создания приложений, работающих с базами данных. А сервер Word дополнит ваше приложение возможностями включения информации в любые отчеты, обзоры, аналитические записки, в которых должна фигурировать информация из баз данных.

Часть IV

Справочные сведения

- Глава 12 Справочные данные по языку C++
- Глава 13 Типы данных в языке C++
- Глава 14 Справочные данные по интегрированной среде разработки C++Builder
- Глава 15 Функции C, C++, библиотек C++Builder, API Windows
- Глава 16 Свойства, методы, события, типы, классы



Справочные данные по языку C++

В настоящей главе приводятся основные справочные сведения по той версии языка C++, которая используется в C++Builder. Впрочем, некоторые конструкции, применяемые в C++Builder, характерны скорее для языка C, а не C++. А некоторые особенности языка, связанные с библиотечными компонентами, относятся к языку Object Pascal. Так что сведения, приводимые в этой и последующих главах, относятся ко всем языкам, используемым в C++Builder. Впрочем, в случаях, когда возможно применить несколько альтернативных подходов, предпочтение все-таки отдается C++.

12.1 Синтаксис языка

Основные синтаксические правила записи программ на языке C++ сводятся к следующему:

- Прописные и строчные буквы считаются разными символами. Поэтому, например, идентификаторы **DATABASE**, **DataBase**, **Database** и **database** относятся к совершенно разным переменным, константам или объектам. При записи идентификаторов могут использоваться латинские буквы, цифры, символ подчеркивания «_». Идентификатор не может начинаться с цифры и не может содержать пробельных символов. Длина идентификатора не ограничена, но ради удобства чтения программы надо стремиться использовать короткие и осмысленные идентификаторы.
- Пробельные символы (пробелы, знаки табуляции, символ новой строки, комментарий) могут размещаться в любом месте текста, но не внутри идентификатора.
- Комментарии в тексте заключаются в скобки вида `/* текст комментария */`. Такие комментарии могут вводиться в любом месте текста, в частности, внутри операторов, и занимать любое количество строк. Вложенные комментарии обычно не допускаются. Считается, что комментарий закончился, как только в тексте встретились первые символы `*/`. Впрочем, в C++Builder 5 можно обеспечить использование вложенных комментариев. Для этого надо включить опцию *Nested Comments* на странице *Advanced Compiler* окна опций проекта. Однако, в стандарте C вложенные комментарии не допускаются, так что их использование делает код непереносимым на другие платформы. Именно поэтому данная опция по умолчанию выключена. Еще один способ введение комментария — размещение его после двух символов слеш `/**`. Этот комментарий должен занимать конец строки, в которой он введен, и не может переходить на следующую строку. Любой текст в строке, помещенный после символов `/**`, воспринимается как комментарий.
- Каждое предложение языка кончается символом точка с запятой «;». Немногие исключения из этого правила будут оговорены особо.
- В строке может размещаться несколько операторов. Однако, с точки зрения простоты чтения текста этим не надо злоупотреблять. Вообще, надо писать программу так, чтобы ее было легко читать и вам, и постороннему человеку,

которому, может быть, придется ее сопровождать. Надо выделять объединенные смыслом операторы в группы, широко используя для этого отступы и комментарии.

- Фигурные скобки { } выделяют составной оператор. Все операторы, помещенные между ними, воспринимаются синтаксически как один оператор.
- Все используемые типы, константы, переменные, функции должны быть объявлены или описаны до их первого использования. Объявления могут встречаться в любом месте текста.

Структуру программы на языке C++ и отдельных ее модулей см. в главе 1 в разделах 1.5.3 и 1.5.4.

12.2 Директивы препроцессора

Обработка программы препроцессором происходит перед ее компиляцией. На этом этапе предварительной обработки вы можете выполнить следующие действия: включить в компилируемый файл другие файлы, определить *символические константы и макросы*, задать режим *условной компиляции* программного кода и *условного выполнения директив препроцессора*. Все директивы препроцессора начинаются с символа «#» и до начала директивы в строке могут находиться только символы пробела. Любая строка, начинающаяся с символа «#», воспринимается как директива препроцессора.

Предупреждение

В конце директив препроцессора не ставится точка с запятой.

12.2.1 Директива #include

Директива **#include** применяется для включения копии указанного в директиве файла в то место, где находится эта директива. Существуют три формы директивы **#include**:

```
#include <имя_файла>
#include "имя_файла"
#include идентификатор макроса
```

Последняя форма предполагает, что первый значащий символ после слова **include** не равен ни '<', ни '''. Предполагается, что макрос, идентификатор которого используется в этой форме директивы, предварительно определен и использует одну из первых двух форм директивы **#include**.

Различие между первыми двумя формами директивы заключается в методе поиска препроцессором включаемого файла. Если имя файла заключено в угловые скобки (< и >), как это делается для включения заголовочных файлов стандартной библиотеки, то последовательность поиска препроцессором заданного файла в каталогах определяется заданными каталогами включения (include directories). Если же имя файла заключено в кавычки, препроцессор ищет файл, просматривая каталоги в следующей последовательности:

- каталог того файла, который содержит директиву **#include**
- каталоги файлов, которые включили в данный файл директивой **#include**
- текущий каталог
- каталоги, указанные опцией компилятора **/I**
- каталоги, заданные переменной окружения **INCLUDE**

Впрочем, этот поиск производится только в случае, если имя файла указано без пути к нему. Если же файл в директиве указан с путем, то никакие другие каталоги не просматриваются.

Обработка директивы **#include** препроцессором сводится к тому, что директива убирается из текста и на ее место заносится копия указанного файла.

Директива **#include** обычно используется для включения стандартных заголовочных файлов библиотек и для включения заголовочных файлов в файлы их реализации. Директива **#include** используется также при работе с программами, состоящими из нескольких исходных файлов, которые должны компилироваться вместе. В C++Builder это соответствует файлам с несколькими формами.

Рассмотрим примеры директив.

Следующая директива включает файл **vcl.h**, который ищется в стандартном каталоге включаемых файлов:

```
#include <vcl.h>
```

Следующая директива включает файл **Unit1.h**, который ищется прежде всего в каталоге, в котором расположен файл, содержащий данную директиву:

```
#include "Unit1.h"
```

Следующие директивы включают файл **C:\Test\My.h**, который ищется только в каталоге **C:\Test**:

```
#define myincl "C:\Test\My.h"  
#include myincl
```

12.2.2 Директивы препроцессора **#define** и **#undef**

12.2.2.1 Символические константы

Директивы препроцессора **#define** создают *символические константы* или макросы без параметров, обозначаемые идентификаторами, и *макросы* — операции, обозначаемые символьными строками. Формат директивы препроцессора **#define** при объявлении символической константы:

```
#define идентификатор_константы замещающий_текст
```

Если замещающий текст длинный, его можно перенести на следующую строку, введя символ обратного слеша «\».

Приведенная форма директивы создает макрос без параметров, называемый обычно символической константой. После появления этой строки в файле все встретившиеся далее в тексте программы имена, совпавшие с элементом директивы идентификатор_константы, будут автоматически заменены на указанный в директиве замещающий_текст прежде, чем начнется компиляция программы. Например, после задания директивы

```
#define PI 3.14159
```

все последующие вхождения в текст программы символической константы **PI** будут заменены на численную константу 3.14159. Замена идентификатора константы не производится в комментариях и строках символов. Если замещающий текст в директиве не задан, то во всем тексте идентификаторы константы просто стираются.

После замены текста этот текст опять просматривается препроцессором в поисках необходимости новых замен. Таким образом, можно использовать вложенные определения символических констант.

Символические константы дают возможность программисту присвоить константе имя и использовать его далее в программе. Если возникнет необходимость изменить значение константы во всей программе, для этого достаточно будет внести только одно изменение в директиву препроцессора **#define** и перекомпилировать программу; значение константы будет изменено по всей программе автоматически.

Предупреждение

Учтите, что все, что находится справа от идентификатора символической константы, является замещающим ее текстом. Например, после выполнения директивы `#define PI=3.14159` пре-процессор заменит все имена `PI` на текст «`=3.14159`».

В C++ отдается предпочтение использованию именованных переменных типа `const`, а не символических констант. Константные переменные являются данными определенного типа и их имена видны отладчику. А если используется символическая константа, то после того, как символическая константа была заменена на соответствующий текст, только этот текст и будет виден отладчику. Правда, недостатком переменных типа `const` является то, что им требуется память в объеме, соответствующем их типу, для хранения своего значения, тогда как для символических констант не требуется никакой дополнительной памяти.

Ниже приведены примеры определения с помощью директивы `#define` символических констант:

```
// определение строки текста:
#define Anyk "Нажмите любую клавишу"

// идентификатор Delete в тексте просто удалится:
#define Delete

// определение директивы #include:
#define GETSTD #include <stdio.h>
```

12.2.2.2 Макросы с параметрами

Формат директивы `#define`, определяющей макрос с параметрами:

```
#define идентификатор_макроса (аргументы) замещающий_текст
```

Между идентификатором макроса и открывающей скобкой не должно быть пробела.

Вызов макроса осуществляется выражением:

```
идентификатор_макроса (аргументы)
```

Макрос, определяемый директивой препроцессора `#define`, это символическое имя некоторых операций. Как и в случае символических констант, идентификатор макроса заменяется на замещающий текст до начала компиляции программы. Но сначала в замещающий текст подставляются значения параметров, а затем уже этот расширенный макрос подставляется в текст вместо идентификатора макроса и списка его параметров.

Например, следующий макрос с одним параметром определяет площадь круга, воспринимая передаваемый в него параметр как радиус:

```
#define CIRC(x) (3.14159 * (x) * (x))
```

Везде в тексте файла, где появится идентификатор `CIRC (A)`, значение аргумента `A` будет использовано для замены `x` в замещающем тексте и этот расширенный текст макроса будет использован для замещения. Например, оператор с макросом в тексте программы

```
S = CIRC(4);
```

примет вид:

```
S = (3.14159 * (4) * (4));
```

Поскольку это выражение состоит только из констант, его значение будет вычислено во время компиляции и полученный результат будет присвоен переменной `S` во время выполнения программы.

Если вызов имеет вид

```
S = CIRC(a + b);
```

то после расширения макроса текст будет иметь вид:

```
S = (3.14159 * (a + b) * (a + b));
```

В данном случае аргумент макроса является выражением, содержащим переменные **a** и **b**. Поэтому вычисления будут осуществляться не во время компиляции, а во время выполнения программы.

Обратите внимание на круглые скобки вокруг каждого включения параметра **x** в тексте рассмотренного макроса и вокруг всего выражения. При вызове типа **CIRC(4)** они кажутся излишними. Но во втором примере вызова при отсутствии скобок расширение привело бы к оператору:

```
S = 3.14159 * a + b * a + b;
```

Тогда в соответствии со старшинством операций (см. раздел 12.7.15) сначала выполнялось бы умножение **3.14159 * a**, затем **b * a**, а затем результаты этих умножений сложились бы друг с другом и с **b**. Конечно, результат вычислений был бы неверным.

Хороший стиль программирования

При объявлении макроса заключайте в скобки параметры в замещающем тексте и сам замещающий текст. Это избавит от возможных неприятностей, связанных с неверной последовательностью вычислений при расширении макроса.

Приведем еще один пример: макрос, определяющий площадь эллипса через значения его полуосей, может быть объявлен директивой

```
#define Ell(x,y) (3.14159 * (x) * (y))
```

Вызов этого макроса может иметь вид:

```
S = Ell(R1, R2);
```

С точки зрения получаемых результатов вычислений макросы эквивалентны функциям. Например, вычисление площади круга можно было бы оформить функцией:

```
double circ(double x)
{
    return 3.14159 * x * x;
}
```

и вызывать ее оператором:

```
S = circ(a + b);
```

Таким образом, возникает вопрос, что выгоднее использовать: макросы или функции.

Вызов функции сопряжен с накладными расходами и затягивает выполнение программы. Это соображение работает в пользу использования макросов. С другой стороны, макрос расширяется во всех местах текста, где используется его вызов. Если таких мест в программе много, то это увеличивает размер текста и, соответственно, размер выполняемого модуля. Так что функции позволяют сокращать объем выполняемого файла, а макросы — сокращать скорость выполнения. Правда, макросы тоже могут быть связаны с дополнительными накладными расходами. В приведенном примере значение параметра **a + b** вычисляется дважды, в то время, как в функции это вычисление осуществляется только один раз. Конечно, для таких простых вычислений это не существенно. Но если в качестве параметра передается сложное выражение, обращающееся в свою очередь к каким-нибудь сложным функциям, то эти дополнительные накладные расходы могут стать заметными и затянуть вычисления.

Недостатком макросов является отсутствие встроенного контроля согласования типов аргументов и формальных параметров. Отсутствие соответствующих предупреждений компилятора может приводить к ошибкам программы, которые трудно отлавливать. Но наиболее существенный недостаток макросов — возможность появления побочных эффектов, если в качестве аргумента в макрос передается некоторое выражение. Например, если описанный выше макрос **CIRC**, вычисляющий площадь круга, вызвать следующим образом:

```
S = CIRC(a++)
```

предполагая рассчитать площадь и затем операцией постфиксного инкремента (см. раздел 12.7.2) увеличить радиус на 1, то макрос будет расширен так:

```
S = (3.14159 * (a++) * (a++));
```

При этом площадь будет вычислена верно, но постфиксный инкремент вычислится два раза. В результате значение радиуса **a** будет увеличено не на 1, а на 2.

Если же это макрос вызвать следующим образом:

```
S = CIRC(++a)
```

предполагая увеличить радиус на 1 и вычислить площадь круга с таким увеличенным радиусом, то макрос будет расширен так:

```
S = (3.14159 * (++a) * (++a));
```

При этом площадь будет определена неверно, так как в процессе вычислений радиус будет увеличен дважды и выражение окажется эквивалентным следующему:

```
S = (3.14159 * (a + 1) * (a + 2));
```

Всех этих побочных эффектов не будет, если вместо макроса использовать описанную выше функцию **circ**.

При выборе реализации вычислений функцией или макросом надо обеспечить компромисс между скоростью вычислений и затратами памяти. Для небольших функций, возможно, наилучшим решением является применение встраиваемых функций (**inline** — см. раздел 12.5.6). Для них проблемы оптимальной реализации решает компилятор, и делает это он, вероятно, не хуже нас с вами.

Хороший стиль программирования

Избегайте применения сложных макросов с параметрами, так как они могут приводить к нежелательным побочным эффектам. Вместо подобных макросов лучше использовать встраиваемые функции **inline**.

12.2.2.3 Директива **#undef**

Определения символических констант и макросов могут быть аннулированы при помощи директивы препроцессора **#undef**, имеющей вид:

```
#undef идентификатор
```

Директива отменяет определение символической константы или макроса с указанным идентификатором. Таким образом, область действия символической константы или макроса начинается с места их определения и заканчивается явным их аннулированием директивой **#undef** или концом файла. После аннулирования соответствующий идентификатор может быть снова использован в директиве **#define**.

Например, возможен следующий код:

```
#define MyConst 128
// Здесь константа MyConst равна 128
...
```

```
#undef MyConst
// Здесь константу MyConst использовать нельзя
...
#define MyConst 64
// Здесь константа MyConst равна 64
...
```

12.2.3 Условная компиляция: директивы `#if`, `#endif`, `#ifdef`, `#ifndef`, `#else`, `#elif`

Условная компиляция дает возможность программисту управлять выполнением директив препроцессора и компиляцией программного кода. Каждая условная директива препроцессора вычисляет значение целочисленного константного выражения. Операции преобразования типов, операция `sizeof` и константы перечислимого типа не могут участвовать в выражениях, вычисляемых в директивах препроцессора.

Условная директива препроцессора `#if` во многом похожа на оператор `if`. Ее синтаксис имеет вид:

```
#if условие
    фрагмент кода
#endif
```

В этой записи условие является целочисленным выражением. Если это выражение возвращает не нуль (истинно), то фрагмент кода, заключенный между директивой `#if` и директивой `#endif`, компилируется. Если же выражение возвращает нуль (ложно), то этот фрагмент игнорируется и препроцессором, и компилятором.

В условиях, помимо обычных выражений, можно использовать конструкцию `defined` идентификатор

`defined` возвращает 1, если указанный идентификатор ранее был определен директивой `#define`, и возвращает 0 в противном случае. Например, возможен следующий код:

```
#if defined Debug && !defined MyConst
    фрагмент кода
#endif
```

Фрагмент кода будет выполняться, если ранее была записана директива

```
#define Debug
```

и не было директивы

```
#define MyConst
```

или эта директива была отменена директивой

```
#undef MyConst
```

Конструкция `#if defined` может быть заменена эквивалентной ей директивой `#ifdef`, а конструкция `#if !defined` — директивой `#ifndef`. Например, тексты

```
#ifdef Size
...
#endif
```

и

```
#if defined Size
...
#endif
```

эквивалентны.

Можно использовать более сложные конструкции условных директив препроцессора при помощи директив **#elif** (эквивалент **else if** в обычной структуре **if**) и **#else** (эквивалент **else** в структуре **if**). Например, в коде

```
#if условие 1
    фрагмент кода 1
#elif условие 2
    фрагмент кода 2
#else
    фрагмент кода 3
#endif
```

фрагмент кода 1 будет компилироваться, если выполняется условие 1, фрагмент кода 2 будет компилироваться, если выполняется условие 2, а фрагмент кода 3 будет компилироваться, если не выполняется ни одно из предыдущих условий.

Условная компиляция может быть полезна во многих случаях. Например, нередко в процессе отладки приложения в него полезно ввести различные отладочные печати, позволяющие следить за ходом выполнения программы (см. главу 2 раздел 2.6.11). Если вы не хотите, чтобы эти печати оставались в окончательном варианте программы, вы можете в разных местах приложения ввести конструкции вида

```
#ifdef Debug
операторы отладки
#endif
```

Тогда, если в начале программы вы введете директиву

```
#define Debug
```

операторы отладки будут компилироваться и выполняться. Но когда вы уберете или закомментируете эту директиву **#define**, определяющую введенный вами идентификатор **Debug**, все операторы отладки исчезнут из текста. Можно поступить даже проще, ничего не изменяя в тексте, а оперируя опцией **Conditionals** на странице **Directories/Conditionals** диалогового окна **Project Options** (см. в главе 14 раздел 14.2.8).

Конечно, вы могли бы поступить иначе: ввести переменную булева типа **Debug**, задать ей в начале выполнения приложения значение **true** и оформлять отладки следующим образом:

```
#if (Debug)
{
    операторы отладки
}
```

Если в дальнейшем заменить задаваемое значение **Debug** на **false**, то операторы отладки перестанут выполняться. Отличие этого подхода от использования директив препроцессора заключается в том, что коды операторов отладки в этом случае останутся в тексте программы, увеличивая размер выполняемого модуля. А директивы условной компиляции просто уберут отладочный код из программы.

Приведем еще один пример использования условной компиляции. Если вы взглянете на заголовочный файл любого модуля формы вашего приложения, то увидите, что **C++Builder** первыми операторами вставляет в него директивы вида:

```
ifndef Unit1H
define Unit1H
```

А завершается заголовочный файл директивой

```
#endif
```

Что это дает? Это позволяет исключить заикливание при циклических директивах **#include**, включающих в различных модулях заголовочные файлы друг друга. Когда в приложение первый раз включается модуль **Unit1.h**, то выполняют-

ся указанные выше первые две директивы и идентификатор **Unit1H** оказывается определен. После этого компилируется текст файла. Но если в результате директив **#include** этот же файл будет включаться еще один раз, то обнаружится, что идентификатор **Unit1H** уже определен, и повторной компиляции файла не произойдет.

12.2.4 Директивы **#error**, **#line**, **#pragma**

Директива препроцессора **#error** имеет следующий синтаксис:

```
#error errmsg
```

Директива печатает в процессе компиляции сообщение об ошибке вида:

```
Error: filename line# : Error directive: errmsg
```

где **errmsg** — сообщение, заданное директивой **#error**. После печати этого сообщения компиляция прекращается.

Директива используется в сочетании с директивами условной компиляции и срабатывает при возникновении условий, не позволяющих продолжить работу. Например:

```
#ifndef Unit1H
#error Не найден файл Unit1.h
```

Директива препроцессора **#line** задает целочисленное константное начальное значение номера строки для нумерации следующих за директивой строк исходного текста программы. Возможны две формы директивы:

```
#line номер_строки
#line номер_строки "имя_файла"
```

Элемент директивы **номер_строки** задает начальное значение номера строки. Все последующие строки исходного текста программы будут нумероваться, начиная с этого номера. Если в директиву включено имя файла, то не только изменяется нумерация последующих строк программы, но и компилятор во всех своих сообщениях будет ссылаться на файл с указанным именем. Директива **#line** обычно используется для того, чтобы сделать сообщения о синтаксических ошибках и предупреждения компилятора более удобными для понимания. Номера строк не добавляются в исходный файл. Пример директивы:

```
#line 100 "Unit1.cpp"
```

Применение директивы **#line** делает работу с отладчиком C++Builder не очень удобной. При возникновении ошибки курсор в окне Редактора Кода останавливается не на строке с ошибкой, а на начале текущего файла или, если в директиве указано имя другого существующего файла, то на начале этого файла. Так что можно рекомендовать не использовать без особой надобности директиву **#line**.

Директива **#pragma** имеет следующий синтаксис:

```
#pragma имя опции
```

и вызывает действия, зависящие от указанной опции. Список возможных опций вы можете найти во встроенной справке C++Builder. Он довольно обширен и связан с различными режимами работы препроцессора.

Пример директивы **#pragma** вы можете видеть в любом модуле своего проекта. Первые две строки файла любого модуля имеют вид:

```
#include <vcl.h>
#pragma hdrstop
```

Здесь использована опция **hdrstop**. Она связана с особенностью работы препроцессора, производительность которого существенно повышается, если учитывается, что некоторое количество заголовочных файлов общие для всех модулей.

Директива **#pragma hdrstop** указывает компилятору конец списка таких общих файлов. Так что надо следить за тем, чтобы не добавлять перед этой директивой включение каких-то заголовочных файлов, не являющихся общими для других модулей.

В файлах модулей вы можете увидеть еще две директивы **#pragma**:

```
#pragma package(smart_init)
#pragma resource "*.dfm"
```

Первая из них определяет последовательность инициализации пакетов такой, какая устанавливается взаимными ссылками использующих их модулей. Вторая говорит препроцессору, что для формы надо использовать файл **.dfm** с тем же именем, что и имя данного файла. Во избежание всяких неприятностей лучше не трогать и не изменять эти директивы.

12.2.5 Операции препроцессора # и

Операция препроцессора **#** применяется к параметрам макросов, представляющим собой лексемы (текст). Операция преобразует лексему в строку символов, взятую в кавычки. Например, если определен следующий макрос:

```
#define Pers(x) Labell->Caption = "Сотрудник " #x
```

и в тексте программы он вызван оператором

```
Pers(Иванов);
```

то он будет расширяться до

```
Labell->Caption = "Сотрудник " "Иванов"
```

Строка «Иванов» заменила параметр **#x** в замещающем тексте. Строки, разделенные символами пробела, сцепляются (склеиваются) во время предварительной обработки, так что вышеприведенный оператор эквивалентен оператору

```
Labell->Caption = "Сотрудник Иванов"
```

Операция **##** выполняет конкатенацию (сцепление, склеивание) двух лексем. Например, если определен макрос

```
#define Concat(x,y) x ## y
```

то встреченное в тексте программы выражение **Concat(Edit,1)** будет преобразовано в **Edit1**.

12.3 Константы

12.3.1 Неименованные константы

Константы могут использоваться непосредственно в тексте программы в любых операторах и выражениях. Имеется 4 типа констант: целые, с плавающей запятой, символьные (включая строки) и перечислимые. Например: 25 и -5 — целые константы, 4.8, 5e15, 5E15, -5.1e8 — константы с плавающей запятой, 'A', '\0', '\n', '007' — символьные константы, «Это строка» — строковая константа.

Целые константы могут быть десятичные, восьмеричные и шестнадцатеричные. Восьмеричные начинаются с символа нуля, после которого следуют восьмеричные цифры (от 0 до 7). Например: 032. Запись константы вида 08 будет воспринята как ошибка, поскольку 8 не является восьмеричной цифрой. Восьмеричные константы не могут превышать значения 03777777777. Значения, большие этой величины, усекаются.

Шестнадцатеричные константы начинаются с символов нуля и X или x, после которых следуют шестнадцатеричные цифры (от 0 до F, можно записывать в верхнем или нижнем регистрах). Например: 0XF01. Шестнадцатеричные константы не могут превышать значения 0xFFFFFFFF. Значения, большие этой величины, усекаются.

Символьные константы должны заключаться в одинарные кавычки. Эти константы хранятся как **char**, **signed char** или **unsigned char**.

Строковые константы заключаются в двойные кавычки. Они хранятся как последовательность символов, завершающаяся нулевым символом '\0'. Пустая строка содержит только нулевой символ.

Если две строковые константы разделены в тексте только пробельным символом, они склеиваются в одну строку. Например:

```
"Это начало строки, " "а это ее продолжение"
```

или

```
"Это начало строки, "  
"а это ее продолжение"
```

воспримутся как константа

```
"Это начало строки, а это ее продолжение"
```

Перенос длинной строки с одной строчки кода в другую можно делать не только так, как показано выше, но и помещая в конец первой строчки одиночный символ обратного слеша '\'. Например, запись

```
"Это начало строки, \  
а это ее продолжение"
```

воспримется как одна строка.

В строковой константе можно использовать управляющие символы, предваряемые обратным слешем (см. раздел 15.1.3 главы 15). Например, константа

```
"\Имя\\"\t\tАдрес\nИванов\t\tМосква"
```

будет при отображении на экране выглядеть так

```
"Имя"           Адрес  
Иванов         Москва
```

Кавычки после символа \ воспринимаются как символ кавычек, а не как окончание строки. Символы \t и \n означают соответственно табуляцию и перевод строки.

Если в константу должен быть включен обратный слеш «\», то надо поместить подряд два слеша. Например, строка

```
"c:\\test\\test.cpp"
```

будет откомпилирована (если компиляция осуществляется с опцией -A) как

```
"c:\test\test.cpp"
```

Предупреждение

В константах, содержащих путь к файлу, не забывайте, что для включения в строку символа '\' надо повторить этот символ два раза.

Константы перечислимого типа объявляются следующим образом:

```
enum имя {значения};
```

Например, оператор

```
enum color { red, yellow, green };
```


объявляет переменную с именем **color**, которая может принимать константные значения **red**, **yellow** или **green**. Эти значения в дальнейшем можно использовать как константы для присваивания переменной **color** или для проверки ее значения. Этим константам соответствуют целые значения, определяемые их местом в списке объявления: **red** — 0, **yellow** — 1, **green** — 2. Эти значения можно изменить, если инициализировать константы явным образом. Например, объявление

```
enum color { red, yellow = 3, green = red + 1};
```

приведет к тому, что значения констант будут равны: **red** — 0, **yellow** — 3, **green** — 1. При этом не обязательно должна соблюдаться уникальность значений. Несколько констант в списке могут иметь одинаковые значения.

В C++Builder имеется ряд предопределенных констант, основные из которых **true** — истина, **false** — ложь, **NULL** — нулевой указатель.

12.3.2 Именованные константы

Именованная константа — это константа, которой присвоен некоторый идентификатор. Объявление именованной константы является указателем для компилятора заменить во всем тексте этот идентификатор значением константы. Такая замена производится только в процессе компиляции и не отражается на исходном тексте.

Цель объявления именованной константы — сделать текст более осмысленным и облегчить при необходимости изменение значения константы во всем тексте. Например, если в тексте многократно используется число 55, означающее максимально допустимое значение каких-то переменных, то проверки

```
if (N > NMax) ...
```

более понятны, чем

```
if (N > 55) ...
```

При необходимости сменить это число, проще изменить его в одном месте программы — в объявлении константы **NMax**, чем искать по всему тексту числа 55, которые, к тому же, в разных частях программы могут иметь разный смысл.

Именованные константы объявляются так же, как переменные (см. раздел 12.4.1), но с добавлением модификатора **const**:

```
const тип имя_константы = значение;
```

Например:

```
const float Pi = 3.14159;
```

В качестве значения константы можно указывать и константное выражение, содержащее ранее объявленные константы. Например, если вы объявили константу **Pi**, то далее можете объявить константы

```
const float Pi2 = 2 * Pi; // удвоенное число Пи
const float Kd = Pi/180; // коэффициент пересчета градусов
                        // в радианы
```

Для целых констант тип можно не указывать:

```
const maxint = 12345;
```

Предупреждение

Не забывайте указывать тип для констант, тип которых отличен от **int**. Например, объявление `const Pi = 3.14159;` присвоит константе **Pi** значение 3, поскольку константа без указания типа считается целой.

Попытка где-то в тексте изменить значение именованной константы приведет к ошибке компиляции с выдачей соответствующего сообщения.

Приведем еще примеры объявления именованных констант:

```
char *const str1 = "Привет!";  
char const *str2 = "Всем привет!";
```

Первое объявление вводит константу **str1**, являющуюся постоянным указателем на строку. Второе объявляет указатель **str2** на строковую константу. Этот указатель не является константой. Его в процессе выполнения программы можно изменить так, чтобы он указывал на другую строковую константу. Иначе говоря, оператор

```
str2 = str1;
```

допустим, а оператор

```
str1 = str2;
```

вызовет ошибку компиляции.

12.4 Переменные

12.4.1 Объявление переменных

Переменная является идентификатором, обозначающим некоторую область в памяти, в которой хранится значение переменной. Это значение может изменяться во время выполнения приложения.

Объявление переменной имеет вид:

```
тип список_идентификаторов_переменных;
```

Список идентификаторов может состоять из идентификаторов переменных, разделенных запятыми. Например:

```
int x1, x2;
```

Одновременно с объявлением некоторые или все переменные могут быть инициализированы. Например:

```
int x1 = 1, x2 = 2;
```

Для инициализации можно использовать не только константы, но и произвольные выражения, содержащие объявленные ранее константы и переменные. Например:

```
int x1 = 1, x2 = 2 * x1;
```

Объявление переменных может быть отдельным оператором или делаться внутри таких операторов, как, например, оператор цикла:

```
for ( int i = 0; i < 10; i++)
```

12.4.2 Классы памяти

Каждая переменная характеризуется некоторым *классом памяти*, который определяет ее *время жизни* — период, в течение которого эта переменная существует в памяти. Одни переменные существуют недолго, другие — неоднократно создаются и уничтожаются, третьи — существуют на протяжении всего времени выполнения программы.

В C++Builder имеется четыре спецификации класса памяти: **auto**, **register**, **extern** и **static**. Спецификация класса памяти идентификатора определяет его класс памяти, область действия и пространство имен.

Областью действия (областью видимости) идентификатора называется область программы, в которой на данную переменную (как, впрочем, и на любой идентификатор — константу, функцию и т.п.) можно сослаться. На некоторые переменные можно сослаться в любом месте программы, тогда как на другие — только в определенных ее частях.

Класс памяти определяется, в частности, местом объявления переменной. *Локальные переменные* объявляются внутри некоторого блока или функции. Эти переменные видны только в пределах того блока, в котором они объявлены. *Блоком* называется фрагмент кода, ограниченный фигурными скобками «{ }». *Глобальные переменные* объявляются вне какого-либо блока или функции.

Спецификации класса памяти могут быть разбиты на два класса: *автоматический класс памяти с локальным временем жизни* и *статический класс памяти с глобальным временем жизни*. Ключевые слова **auto** и **register** используются для объявления переменных с локальным временем жизни. Эти спецификации применимы только к локальным переменным. Локальные переменные создаются при входе в блок, в котором они объявлены, существуют лишь во время активности блока и исчезают при выходе из блока.

Спецификация **auto**, как и другие спецификации, может указываться перед типом в объявлении переменных. Например:

```
auto float x, y;
```

Локальные переменные являются переменными с локальным временем жизни по умолчанию, так что ключевое слово **auto** используется редко. Далее мы будем ссылаться на переменные автоматического класса памяти просто как на автоматические переменные.

Пусть, например, имеется следующий фрагмент кода:

```
{  
    int i = 1;  
    ...  
    i++;  
    ...  
}
```

к которому в ходе работы программы происходит неоднократное обращение. При каждом таком обращении переменная *i* будет создаваться заново (под нее будет выделяться память) и будет инициализироваться единицей. Затем в ходе работы программы ее значение будет увеличиваться на 1 операцией инкремента. В конце выполнения этого блока переменная исчезнет и выделенная под нее память освободится. Следовательно, в такой локальной переменной невозможно хранить какую-то информацию между двумя обращениями к блоку.

Спецификация класса памяти **register** может быть помещена перед объявлением автоматической переменной, чтобы компилятор сохранял переменную не в памяти, а в одном из высокоскоростных аппаратных регистров компьютера. Например:

```
register int i = 1;
```

Если интенсивно используемые переменные, такие как счетчики или суммы могут сохраняться в аппаратных регистрах, накладные расходы на повторную загрузку переменных из памяти в регистр и обратную загрузку результата в память могут быть исключены. Это сокращает время вычислений.

Компилятор может проигнорировать объявления **register**. Например, может оказаться недостаточным количество регистров, доступных компилятору для использования. К тому же оптимизирующий компилятор способен распознавать часто используемые переменные и решать, помещать их в регистры или нет. Так что явное объявление спецификации **register** используется редко.

Ключевые слова **extern** и **static** используются, чтобы объявить идентификаторы переменных как идентификаторы статического класса памяти с глобальным временем жизни. Такие переменные существуют с момента начала выполнения программы. Для таких переменных память выделяется и инициализируется сразу после начала выполнения программы.

Существует два типа переменных статического класса памяти: глобальные переменные и локальные переменные, объявленные спецификацией класса памяти **static**. Глобальные переменные по умолчанию относятся к классу памяти **extern**. Глобальные переменные создаются путем размещения их объявлений вне описания какой-либо функции и сохраняют свои значения в течение всего времени выполнения программы. На глобальные переменные может ссылаться любая функция, которая расположена после их объявления или описания в файле.

Хороший стиль программирования

Переменные, используемые только в отдельной функции, предпочтительнее объявлять как локальные переменные этой функции, а не как глобальные переменные. Это облегчает чтение программы и позволяет избежать случайного доступа к таким переменным других функций.

Локальные переменные, объявленные с ключевым словом **static**, известны только в том блоке, в котором они определены. Но в отличие от автоматических переменных, локальные переменные **static** сохраняют свои значения в течение всего времени выполнения программы. При каждом следующем обращении к этому блоку локальные переменные содержат те значения, которые они имели при предыдущем обращении.

Вернемся к уже рассмотренному выше примеру, но укажем для переменной **i** статический класс:

```
{
    static int i = 1;
    ...
    i++;
    ...
}
```

Инициализация переменной **i** произойдет только один раз за время выполнения программы. При первом обращении к этому блоку значение переменной **i** будет равно 1. К концу выполнения блока ее значение станет равно 2. При следующем обращении к блоку это значение сохранится и при окончании повторного выполнения блока **i** будет равно 3. Таким образом, статическая переменная способна хранить информацию между обращениями к блоку и, следовательно, может использоваться, например, как счетчик числа обращений.

Все числовые переменные статического класса памяти принимают нулевые начальные значения, если программист явно не указал другие начальные значения. Статические переменные — указатели, тоже имеют нулевые начальные значения.

Спецификации класса памяти **extern** используются в программах с несколькими файлами. Пусть, например, в модуле **Unit1** в файле **Unit1.cpp** или **Unit1.h** (это безразлично) объявлена глобальная переменная

```
int a = 5;
```

Тогда, если в другом модуле **Unit2** в файле **Unit2.cpp** или **Unit2.h** объявлена глобальная переменная

```
extern int a;
```

то компилятор понимает, что речь идет об одной и той же переменной. И оба модуля могут с ней работать. Для этого даже нет необходимости связывать эти модули директивой **#include** (см. раздел 12.2.1), включающей в модуль **Unit1** заголовочный файл второго модуля.

Подробнее области видимости переменных рассмотрены в разделе 12.6.

12.5 Функции

12.5.1 Объявление и описание функций

Функции представляют собой программные блоки, которые могут вызываться из разных частей программы. При вызове в них передаются некоторые переменные, константы, выражения, являющиеся аргументами, которые в самих процедурах и функциях воспринимаются как формальные параметры. При этом функции возвращают значение определенного типа, которое замещает в вызвавшем выражении имя вызванной функции.

Например, оператор

```
I = 5 * F(X);
```

вызывает функцию **F** с аргументом **X**, умножает возвращенное ею значение на 5 и присваивает результат переменной **I**.

Допускается также вызов функции, не использующий возвращаемого ею значения. Например:

```
F(X);
```

В этом случае возвращаемое функцией значение игнорируется.

Функция описывается следующим образом:

```
тип_возвращаемого_значения имя_функции(список_параметров)
{
    операторы тела функции
}
```

Первая строка этого *описания*, содержащая тип возвращаемого значения, имя функции и список параметров, называется *заголовком* функции. Тип возвращаемого значения может быть любым, кроме массива и функции. Могут быть также функции, не возвращающие никакого значения. В заголовке таких функций тип возвращаемого значения объявляется **void**.

Если тип возвращаемого значения не указан, он по умолчанию считается равным **int**.

Хороший стиль программирования

Хотя тип возвращаемого значения **int** можно не указывать в заголовке функции, не следует использовать эту возможность. Всегда указывайте тип возвращаемого значения, кроме главной функции **main**. Указание типа делает программу более наглядной и предотвращает возможные ошибки, связанные с неправильным преобразованием типов.

Список параметров, заключаемый в скобки, в простейшем случае (более сложные формы задания списка параметров будут рассмотрены позднее) представляет собой разделяемый запятыми список вида

```
тип_параметра идентификатор_параметра
```

Например, заголовок:

```
double FSum(double X1, double X2, int A)
```

объявляет функцию с именем **FSum**, с тремя параметрами **X1**, **X2** и **A**, из которых первые два имеют тип **double**, а последний — **int**. Тип возвращаемого результата — **double**. Имена параметров **X1**, **X2** и **A** — локальные, т.е. они имеют значение только внутри данной функции и никак не связаны с именами аргументов, переданных при вызове функции. Значения этих параметров в начале выполнения функции равны значениям аргументов на момент вызова функции. Подробнее эти вопросы будут рассмотрены в разделе 12.5.2.

Ниже приведен заголовок функции, не возвращающей никакого значения:

```
void SPrint(AnsiString S)
```

Она принимает один параметр типа строки и, например, отображает его в каком-нибудь окне приложения.

Если функция не принимает никаких параметров, то скобки или оставляются пустыми, или в них записывается ключевое слово **void**. Например:

```
void Fl(void)
```

или

```
void Fl()
```

Хороший стиль программирования

Всегда указывайте **void** в списке параметров, если функция не получает никаких параметров. Это делает программу более переносимой.

Предупреждение

Роль пустого списка параметров функции в C++ существенно отличается от аналогичного списка в языке C. В C это означает, что все проверки аргументов отсутствуют (т.е. вызов функции может передать любой аргумент, который требуется). А в C++ пустой список означает отсутствие аргументов. Таким образом, программа на C, использующая эту особенность, может сообщить о синтаксической ошибке при компиляции в C++.

Как правило (хотя формально не обязательно), помимо описания функции в текст программы включается также *прототип* функции — ее предварительное объявление. Прототип представляет собой тот же заголовок функции, но с точкой с запятой «;» в конце. Кроме того, в прототипе можно не указывать имена параметров. Если вы все-таки указываете имена, то их область действия является только этот прототип функции. Вы можете использовать те же идентификаторы в любом месте программы в любом качестве. Таким образом, указание имен параметров в прототипе обычно преследует только одну цель — документирование программы, напоминание вам или сопровождающему программу человеку, какой параметр что именно обозначает.

Примеры прототипов приведенных выше заголовков функций:

```
double FSum(double X1, double X2, int A);  
void SPrint(AnsiString S);  
void Fl(void);
```

или

```
double FSum(double, double, int);  
void SPrint(AnsiString);  
void Fl();
```

Введение в программу прототипов функций преследует несколько целей. Во-первых, это позволяет использовать в данном модуле функцию, описанную в каком-нибудь другом модуле. Тогда из прототипа компилятор получает сведения, сколько параметров, какого типа и в какой последовательности получает данная функция. Во-вторых, если в начале модуля вы определили прототипы функций, то последовательность размещения в модуле описания функций безразлична. При отсутствии прототипов любая используемая функция должна быть описана до ее первого вызова в тексте. Это прибавляет вам хлопот, а иногда при взаимных вызовах функций друг из друга вообще невозможно. И, наконец, прототипы, размещенные в одном месте (обычно в начале модуля), делают программу более наглядной и самодокументированной. Особенно в случае, если вы снабжаете прототипы хотя бы краткими комментариями.

Если предполагается, что какие-то из описанных в модуле функций могут использоваться в других модулях, прототипы этих функций следует включать в заголовочный файл. Тогда в модулях, использующих данные функции, достаточно будет написать директиву **#include** (см. раздел 12.2.1), включающую данный заголовочный файл, и не надо будет повторять прототипы функций.

Хороший стиль программирования

Включайте в модуль где-то в одном месте (обычно в начале) прототипы всех описанных в нем ваших функций с краткими комментариями. Это хорошо документирует программу, делает ее нагляднее, позволяет вам не заботиться о последовательности описаний функций. Если вы хотите, чтобы какие-то из описанных в модуле функций могли использовать другие модули, включайте прототипы этих функций в заголовочный файл.

Обычно функции принимают указанное в прототипе число параметров указанных типов. Однако, могут быть функции, принимающие различное число параметров (например, библиотечная функция **printf**) или параметры неопределенных заранее типов. В этом случае в прототипе вместо неизвестного числа параметров или вместо параметров неизвестного типа ставится многоточие «...». Многоточие может помещаться только в конце списка параметров после известного числа параметров известного типа или полностью заменять список параметров. Например:

```
int prf(char *format, ...);
```

Функция с подобным прототипом принимает один параметр **format** типа **char** * (например, строку форматирования) и произвольное число параметров произвольного типа. Функция с прототипом

```
void Fp(...);
```

может принимать произвольное число параметров произвольного типа.

Если в прототипе встречается многоточие, то типы соответствующих параметров и их количество компилятором не проверяются.

Объявлению функции могут предшествовать спецификаторы класса памяти **extern** или **static**. Спецификатор **extern** предполагается по умолчанию, так что записывать его не имеет смысла. К функциям, объявленным как **extern**, можно получить доступ из других модулей программы (см. заключительную часть раздела 12.6.1 и раздел 1.5.5.4 в главе 1). Если же объявить функцию со спецификатором **static**, например

```
static void F(void);
```

то доступ к ней из других модулей невозможен. Это надо использовать в крупных проектах во избежание недоразумений при случайных совпадениях имен функций в различных модулях.

Теперь рассмотрим описание тела функции. Тело функции пишется по тем же правилам, что и любой код программы, и может содержать объявления типов, констант, переменных и любые выполняемые операторы. Не допускается объявление и описание в теле других функций. Таким образом, функции не могут быть вложены друг в друга.

Надо иметь в виду, что все объявления в теле функции носят локальный характер. Объявленные переменные доступны только внутри данной функции. Если их идентификаторы совпадают с идентификаторами каких-то глобальных переменных модуля, то эти внешние переменные становятся невидимыми и недоступными. В этих случаях получить доступ к глобальной переменной можно, поставив перед ее именем два двоеточия «::», т.е. применив унарную операцию разрешения области действия.

Локальные переменные не просто видны только в теле функции, но по умолчанию они и существуют только внутри функции, создаваясь в момент вызова функ-

ции и уничтожаясь в момент выхода из функции. Если требуется этого избежать, соответствующие переменные должны объявляться со спецификацией **static** (подробнее см. в разделе 12.4.2).

Выход из функции может осуществляться следующими способами. Если функция не должна возвращать никакого значения, то выход из нее происходит или по достижении закрывающей ее тело фигурной скобки, или при выполнении оператора **return**. Если же функция должна возвращать некоторое значение, то нормальный выход из нее осуществляется оператором

```
return выражение
```

где выражение должно формировать возвращаемое значение и соответствовать типу, объявленному в заголовке функции.

Например:

```
double FSum(double X1, double X2, int A)
{
    return A * (X1 + X2);
}
```

Ниже приведен пример функции, не возвращающей никакого значения:

```
void SPrint(AnsiString S)
{
    if (S != "")
        ShowMessage(S);
}
```

Здесь возврат из функции происходит по достижении закрывающейся фигурной скобки тела функции. Приведем вариант той же функции, использующий оператор **return**:

```
void SPrint(AnsiString S)
{
    if (S == "") return;
    ShowMessage(S);
}
```

Прервать выполнение функции можно также генерацией какого-то исключения (см. раздел 12.10). Наиболее часто в этих целях используется процедура **Abort**, генерирующая «молчаливое» исключение **EAbort**, не связанное с каким-то сообщением об ошибке. Если в программе не предусмотрен перехват этого исключения, то применение функции **Abort** выводит управление сразу наверх из всех вложенных друг в друга вызовов функций.

Возвращаемое функцией значение может включать в себя вызов каких-то функций. В том числе функция может вызывать и саму себя, т.е. допускается рекурсия. В качестве примера приведем функцию, рекурсивно вычисляющую факториал. Как известно, значение факториала равно $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$, причем считается, что $1! = 1$ и $0! = 1$. Факториал можно вычислить с помощью простого цикла **for** (и это, конечно, проще). Но можно факториал вычислять и с помощью рекуррентного соотношения $n! = n \cdot (n-1)!$. Для иллюстрации рекурсии воспользуемся именно этим соотношением. Тогда функция **factorial** вычисления факториала может быть описана следующим образом:

```
unsigned long factorial (unsigned long n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Если значение параметра **n** равно 0 или 1, то функция возвращает значение 1. В противном случае функция умножает текущее значение **n** на результат, возвращаемый вызовом той же функции **factorial**, но со значением параметра **n**, уменьшенным на единицу. Поскольку при каждом вызове значение параметра уменьшается, рано или поздно оно станет равно 1. После этого цепочка рекурсивных вызовов начнет свертываться и в конце концов вернет значение факториала.

12.5.2 Передача параметров в функции по значению и по ссылке

Список параметров, передаваемый в функции, как было показано в предыдущем разделе, состоит из имен параметров и указаний на их тип. Например, в заголовке

```
double FSum(double X1, double X2, int A)
```

указано три параметра **X1**, **X2**, **A** и определены их типы. Вызов такой процедуры может иметь вид:

```
Pr(Y, X2, 5);
```

Это только один из способов передачи параметров в процедуру, называемый *передачей по значению*. Работает он так. В момент вызова функции в памяти создаются временные переменные с именами **X1**, **X2**, **A**, и в них копируются значения аргументов **Y**, **X2** и константы 5. На этом связь между аргументами и переменными **X1**, **X2**, **A** разрывается. Вы можете изменять внутри процедуры значения **X1**, **X2** и **A**, но это никак не отразится на значениях аргументов. Аргументы при этом надежно защищены от непреднамеренного изменения своих значений вызванной функцией. Это предотвращает случайные побочные эффекты, которые так сильно мешают иногда созданию корректного и надежного программного обеспечения.

К недостаткам такой передачи параметров по значению относятся затраты времени на копирование значений и затраты памяти для хранения копии. Если речь идет о какой-то переменной простого типа, это, конечно, не существенно. Но если, например, аргумент — массив из тысяч элементов, то соображения затрат времени и памяти могут стать существенными.

Еще одним недостатком передачи параметров по значению является невозможность из функций изменять значения некоторых аргументов, что во многих случаях очень желательно.

Возможен и другой способ передачи параметров — *вызов по ссылке*. В случае вызова по ссылке оператор вызова дает вызываемой функции возможность прямого доступа к передаваемым данным, а также возможность изменения этих данных. Вызов по ссылке хорош в смысле производительности, потому что он исключает накладные расходы на копирование больших объемов данных; в то же время он может ослабить защищенность, потому что вызываемая функция может испортить передаваемые в нее данные.

Вызов по ссылке можно осуществить двумя способами: с помощью ссылочных параметров и с помощью указателей. Ссылочный параметр — это псевдоним соответствующего аргумента. Чтобы показать, что параметр функции передан по ссылке, после типа параметра в прототипе функции ставится символ амперсанта (&); такое же обозначение используется в списке типов параметров в заголовке функции. Перед амперсантом и после него могут вставляться пробельные символы. Например, идентичные объявления

```
int &count
int & count
int& count
```

в списке параметров заголовка функции могут читаться как «**count** является ссылкой на **int**». В вызове такой функции достаточно указать имя переменной и она бу-

дет передана по ссылке. Реально в функцию передается не сама переменная, а ее адрес, полученный операцией адресации (&). Тогда упоминание в теле вызываемой функции переменной по имени ее параметра в действительности является обращением к исходной переменной в вызывающей функции и эта исходная переменная может быть изменена непосредственно вызываемой функцией.

Например:

```
void square(int &);           // Прототип функции вычисления квадрата

void square(int &a)           // Заголовок функции
{
    a *= a;                   // Изменение значения параметра
}
```

Вызываться подобная функция может обычным способом передачей в нее имени аргумента. Например:

```
int x1 = 2;
square(x1);
```

В результате подобного вызова переменная **x1** получит значение 4.

Предупреждение

Поскольку ссылочные параметры упоминаются в теле вызываемой функции просто по имени, программист может нечаянно принять ссылочный параметр за параметр, переданный по значению. Это может привести к неприятным ошибкам, если исходные значения переменных изменяются вызывающей функцией.

Альтернативной формой передачи параметра по ссылке является использование указателей (см. раздел 13.7 главы 13). Тогда адрес переменной передается в функцию не операцией адресации (&), а операцией косвенной адресации (*). В списке параметров подобной функции перед именем переменной указывается символ «*», свидетельствуя о том, что передается не сама переменная, а указатель на нее. В теле функции тоже перед именем параметра ставится символ операции разыменования *, чтобы получить доступ через указатель к значению переменной (пояснения всего этого вы можете найти в разделе 13.7 главы 13). А при вызове функции в нее в качестве аргумента должна передаваться не сама переменная, а ее адрес, получаемый с помощью операции адресации &.

Приведем пример той же рассмотренной ранее функции **square**, но с передачей параметра по ссылке с помощью указателя:

```
void square(int *);           // Прототип функции вычисления квадрата

void square(int *a)           // Заголовок функции
{
    *a *= *a;                 // Изменение значения параметра
}
```

Вызов подобной функции может осуществляться, например, следующим образом:

```
int x1 = 2;
square(&x1);
```

12.5.3 Применение при передаче параметров спецификации const

В предыдущем разделе была рассмотрена передача параметров по ссылке. Она решает сразу две задачи: исключает накладные расходы, связанные с копировани-

ем передаваемых значений, и дает функции доступ для изменения значений передаваемых аргументов. Однако, иногда требуется решать только первую задачу: избавиться от копирования громоздких аргументов типа больших массивов. Но при этом не требуется позволять функции изменять значения аргументов.

Это может быть осуществлено передачей в функцию аргументов как констант. Для этого перед соответствующими переменными в списке ставится ключевое слово **const**.

При использовании ссылочного параметра заголовок функции (именно заголовок описания, поскольку в прототипе спецификатор **const** указывать не обязательно) может иметь следующий вид:

```
double F(const &A)
```

В этом случае аргумент **A** не будет копироваться при вызове функции, но внутри функции изменить значение **A** будет невозможно. При попытке сделать такое изменение компилятор выдаст сообщение: «Cannot modify a const object».

Подобная передача параметра как константы позволяет сделать код более эффективным, так как при этом компилятору заведомо известно, что никакие изменения параметра невозможны.

При использовании указателей для передачи параметров в функцию возможны четыре варианта: неконстантный указатель на неконстантные данные, неконстантный указатель на константные данные, константный указатель на неконстантные данные и константный указатель на константные данные. Каждая комбинация обеспечивает доступ с разным уровнем привилегий.

Наивысший уровень доступа предоставляется неконстантным указателем на неконстантные данные — данные можно модифицировать посредством разыменования указателя, а сам указатель может быть модифицирован, чтобы он указывал на другие данные. Это описанная в предыдущем разделе передача параметров по ссылке с помощью указателя. В этом варианте передачи параметров спецификатор **const** не используется.

Неконстантный указатель на константные данные — это указатель, который можно модифицировать, чтобы указывать на любые элементы данных подходящего типа, но сами данные, на которые он ссылается, не могут быть модифицированы. Например, прототип:

```
void F(const char *sPtr);
```

объявляет функцию, в которую передается указатель **sPtr**, указывающий на константные данные типа **const char *** — в данном случае строку (массив символов). В теле функции такой указатель можно менять, перемещая его с одного обрабатываемого символа на другой. Но сами элементы строки (массива) изменять невозможно, так как они объявлены константными. Таким образом, исходные значения сохраняются от их несанкционированного изменения.

Константный указатель на неконстантные данные — это указатель, который всегда указывает на одну и ту же ячейку памяти, данные в которой можно модифицировать посредством указателя. Этот вариант, например, реализуется по умолчанию для имени массива. Имя массива — это константный указатель на начало массива. Используя имя массива и индексы массива можно обращаться ко всем данным в массиве и изменять их. Прототип функции с передачей константного указателя на неконстантные данные может иметь вид:

```
void F(char *const sPtr);
```

Наименьший уровень привилегий доступа предоставляет константный указатель на константные данные. Такой указатель всегда указывает на одну и ту же ячейку памяти и данные в этой ячейке нельзя модифицировать. Это выглядит так, как если бы массив нужно было передать функции, которая только просматривает массив, использует его индексы, но не модифицирует сам массив. Прототип функции с подобной передачей параметра может иметь вид:

```
void F(const char *const sPtr);
```

12.5.4 Параметры со значениями по умолчанию

Обычно при вызове функции в нее передается конкретное значение каждого параметра. Но программист может указать, что параметр является параметром по умолчанию, и приписать этому параметру значение по умолчанию. Делается это заданием в заголовке функции после имени параметра символа «=», после которого записывается значение по умолчанию. Пусть, например, вы хотите написать функцию, которая рассчитывает суммарную силу, действующую на тело объемом V с плотностью P , погруженное в жидкость (например, воду) с плотностью $PH2O$. Как известно, формула, выражающая эту суммарную силу, направленную вверх (если ответ будет отрицательным, значит сила направлена вниз — тело тонет), следующая: $F = G * V * (P - PH2O)$, где G — ускорение свободного падения.

Функцию, определяющую эту силу, можно описать следующим образом:

```
double Arh(double V = 1, double P = 0.5, double PH2O = 1,
double G = 9.81)
{ return G * V * (PH2O - P); }
```

Здесь всем параметрам даны значения по умолчанию. Объем V по умолчанию принят равным 1 м^3 , плотность тела P по умолчанию равна 0.5 т/м^3 (плотность некоторых пород дерева), плотность воды $PH2O$ принята по умолчанию равной 1 т/м^3 , а ускорение свободного падения G принято равным $9,81 \text{ м/с}^2$.

Если при вызове функции параметр по умолчанию не указан, то в функцию автоматически передается его значение по умолчанию. Например, если вызвать приведенную функцию оператором

```
F = Arh();
```

то значение F будет равно силе при значениях всех параметров по умолчанию.

Аргументы по умолчанию должны быть самыми правыми (последними) аргументами в списке параметров функции. Если вызывается функция с двумя или более параметрами по умолчанию и если пропущенный параметр не является самым правым в списке, то все параметры справа от пропущенного тоже пропускаются.

Например, вызов той же функции оператором

```
F = Arh(2);
```

позволяет рассчитать силу, действующую на тело объемом 2 м^3 при значениях всех остальных параметров по умолчанию. Вызов функции оператором

```
F = Arh(2, 2.6);
```

позволяет рассчитать силу, действующую на алюминиевое (плотность 2.6 т/м^3) тело объемом 2 м^3 при значениях остальных параметров по умолчанию. Аналогично, задав при вызове три параметра можно рассчитать силу, действующую на тело, погруженное в жидкость другой плотности, а задав все четыре параметра можно определить силу, действующую на тело при эксперименте, проводящемся не на уровне моря (при этом изменится ускорение свободного падения).

Этот пример показывает, что последними в списке параметров со значениями по умолчанию надо указывать те параметры, значения которых в реальных задачах чаще всего остаются равными заданным по умолчанию.

Пропускать при вызове можно только некоторое число последних параметров в списке. Например, нельзя вызвать функцию таким образом:

```
F = Arh(2, , 1.1); // Ошибочный вызов
```

Параметры по умолчанию должны быть указаны при первом упоминании имени функции — обычно в прототипе. Значения по умолчанию могут быть константами, глобальными переменными или вызовами функций.

12.5.5 Передача в функции переменного числа параметров

Иногда в функции требуется передавать некоторое число фиксированных параметров плюс неопределенное число дополнительных параметров. В этом случае заголовок функции имеет вид:

```
тип имя_функции(список_аргументов, ...)
```

В данном случае список аргументов включает в себя конечное число обязательных аргументов (этот список не может быть пустым), после которого ставится многоточие на месте неопределенного числа параметров. Для работы с этими параметрами в файле **stdarg.h** определен тип списка **va_list** и три макроса: **va_start**, **va_arg** и **va_end**.

Макрос **va_start** имеет синтаксис:

```
void va_start(va_list ap, lastfix)
```

Этот макрос начинает работу со списком, устанавливая его указатель **ap** на первый передаваемый в функцию аргумент из списка с неопределенным числом аргументов. Параметр **lastfix** — это имя последнего из обязательных аргументов функции.

Макрос **va_arg** имеет синтаксис:

```
type va_arg(va_list ap, type)
```

Макрос возвращает значение очередного аргумента из списка. Параметр **type** указывает тип аргумента. Перед вызовом **va_arg** значение **ap** должно быть установлено вызовом **va_start** или **va_arg**. Каждый вызов **va_arg** переводит указатель **ap** на следующий аргумент.

Макрос **va_end** имеет синтаксис:

```
void va_end(va_list ap)
```

Макрос завершает работу со списком, освобождая память. Он должен вызываться после того, как с помощью **va_arg** прочитан весь список аргументов. В противном случае могут быть непредсказуемые последствия.

Рассмотрим пример. Пусть требуется создать функцию **average**, которая рассчитывает и отображает в метке **Label1** среднее значение передаваемых в нее целых положительных чисел. Функция принимает в качестве первого аргумента некоторое сообщение, которое должно отображаться перед результатами расчета. Список обрабатываемых чисел может быть любой длины и заканчиваться нулем. Такая функция может быть реализована следующим образом:

```
#include <stdarg.h>
...
void average(AnsiString mess,...)
{
    double A = 0;
    int i = 0, arg;
    va_list ap;
    va_start(ap, mess);
    while ((arg = va_arg(ap, int)) != 0)
    {
        i++;
        A += arg;
    }
    Form1->Label1->Caption = mess + "N = " + IntToStr(i) +
                                ", среднее = "+FloatToStr(A/i);
    va_end(ap);
}
```

Вызов функции может быть, например, таким:

```
average("Результаты экзамена: ", 4, 2, 3, 5, 4, 0);
```

В результате функция выдаст в метку **Label1** сообщение:

Результаты экзамена: N = 5, среднее = 3,6

Функцию **average** можно было бы организовать иначе, не вводя специальную конечную метку в список (в приведенном примере — 0), а предваряя список аргументов параметром **N**, указывающим размер списка:

```
void average(AnsiString mess,int N,...)
{
    double A = 0;
    va_list ap;
    va_start(ap, N);
    for(int i = 0; i < N; i++)
        A += va_arg(ap,int);
    Form1->Label1->Caption = mess + "N = " +IntToStr(N) +
                                ", среднее = "+FloatToStr(A/N);
    va_end(ap);
}
```

Вызов функции может быть, например, таким:

```
average("Результаты экзамена: ",5,4,2,3,5,4);
```

12.5.6 Встраиваемые функции inline

Реализация программы как набора функций хороша с точки зрения разработки программного обеспечения, но вызовы функций приводят к накладным расходам во время выполнения. В C++ для снижения этих накладных расходов на вызовы функций — особенно небольших функций — предусмотрены встраиваемые (**inline**) функции. Спецификация **inline** перед указанием типа результата в объявлении функции «советует» компилятору сгенерировать копию кода функции в соответствующем месте, чтобы избежать вызова этой функции. Это эквивалентно объявлению соответствующего макроса (см. раздел 12.2.2.2). В результате получается множество копий кода функции, вставленных в программу, вместо единственной копии, которой передается управление при каждом вызове функции.

Компилятор может игнорировать спецификацию **inline**, что обычно и делает для всех функций, кроме самых небольших.

Предупреждение

Любые изменения функции **inline** могут потребовать перекомпиляцию всех «потребителей» этой функции. Это может оказаться существенным моментом для развития и поддержки некоторых программ.

Хороший стиль программирования

Спецификацию **inline** целесообразно применять только для небольших и часто используемых функций. Использование функций **inline** может уменьшить время выполнения программы, но может увеличить ее размер. Применение функций **inline** предпочтительнее объявления макросов, поскольку в данном случае вы даете возможность компилятору оптимизировать код.

Пусть, например, вам во многих частях программы приходится вычислять длину окружности, заданной своим радиусом **R**. Тогда вы можете оформить эти вычисления, определив встраиваемую функцию:

```
inline double Circ(double R){return 6.28318 * R;}
```

Обращение в любом месте программы вида **Circ(2)** приведет к встраиванию в соответствующем месте кода **6.28318 * 2** (если компилятор сочтет это целесообразным).

12.5.7 Перегрузка функций

C++ позволяет определить несколько функций с одним и тем же именем, если эти функции имеют разные наборы параметров (по меньшей мере разные типы параметров). Эта особенность называется перегрузкой функции. При вызове перегруженной функции компилятор C++ определяет соответствующую функцию путем анализа количества, типов и порядка следования аргументов в вызове. Перегрузка функции обычно используется для создания нескольких функций с одинаковым именем, предназначенных для выполнения сходных задач, но с разными типами данных.

Хороший стиль программирования

Перегруженные функции, которые выполняют тесно связанные задачи, делают программы более понятными и легко читаемыми.

Пусть, например, вы хотите определить функции, добавляющие в заданную строку типа (**char ***) символ пробела и значение целого числа, или значение числа с плавающей запятой, или значение булевой переменной. Причем хотите обращаться в любом случае к функции, которую называете, например, **ToS**, предоставив компилятору самому разбираться в типе параметра и в том, какую из функций надо вызывать в действительности. Для решения этой задачи вы можете описать следующие функции:

```
char * ToS(char *S, int X)
{ return strcat(strcat(S, " "), IntToStr(X).c_str()); }

char * ToS(char *S, double X)
{ return strcat(strcat(S, " "), FloatToStr(X).c_str()); }

char * ToS(char *S, bool X)
{ if (X) return strcat(S, " true");
  else return strcat(S, " false"); }
```

Тогда в своей программе вы можете написать, например, вызовы:

```
char S[128] = "Значение =";
char S1 = ToS(S, 5);
```

или

```
char S[128] = "Значение =";
char S2 = ToS(S, 5.3);
```

или

```
char S[128] = "Значение =";
char S3 = ToS(S, true);
```

В первом случае будет вызвана функция с целым аргументом, во втором — с аргументом типа **double**, в третьем — с булевым аргументом. Вы видите, что перегрузив соответствующие функции вы существенно облегчили свою жизнь, избавившись от необходимости думать о типе параметра.

Приведем еще один пример, в котором перегруженные функции различаются количеством параметров. Ниже описана перегрузка функции, названной **Area** и вычисляющей площадь круга по его радиусу **R**, если задан один параметр, и площадь прямоугольника по его сторонам **a** и **b**, если задано два параметра:

```
double Area(double R) { return 6.28318 * R * R; }
double Area(double a, double b) { return a * b; }
```

Тогда операторы вида

```
S1 = Area(1);
S2 = Area(1, 2);
```

приведут в первом случае к вызову функции вычисления площади круга, а во втором — к вызову функции вычисления площади прямоугольника.

Перегруженные функции различаются компилятором с помощью их *сигнатуры* — комбинации имени функции и типов ее параметров. Компилятор кодирует идентификатор каждой функции по числу и типу ее параметров (иногда это называется декорированием имени), чтобы иметь возможность осуществлять надежное связывание типов. Надежное связывание типов гарантирует, что вызывается надлежащая функция и что аргументы согласуются с параметрами. Компилятор выявляет ошибки связывания и выдает сообщения о них.

Для различения функций с одинаковыми именами компилятор использует только списки параметров. Перегруженные функции не обязательно должны иметь одинаковое количество параметров. Программисты должны быть осторожными, имея дело в перегруженных функциях с параметрами по умолчанию, поскольку это может стать причиной неопределенности.

Предупреждение

Функция с пропущенными аргументами по умолчанию может оказаться вызванной аналогично другой перегруженной функции; это синтаксическая ошибка.

Рассмотренный аппарат перегрузки функций — только один из возможных способов решения поставленной задачи, правда, универсальный, позволяющий работать и с разными типами параметров, и с разным числом параметров. В следующем разделе рассмотрен еще один механизм — шаблоны, позволяющий решать аналогичные задачи, правда, для более узких классов функций.

12.5.8 Шаблоны функций

Перегруженные функции обычно используются для выполнения сходных операций над различными типами данных. Если операции идентичны для каждого типа, это можно выполнить более компактно и удобно, используя *шаблоны функций*. Вам достаточно написать одно единственное определение шаблона функции. Основываясь на типах аргументов, указанных в вызовах этой функции, C++ автоматически генерирует разные функции для соответствующей обработки каждого типа. Таким образом, определение единственного шаблона определяет целое семейство решений.

Все определения шаблонов функций начинаются с ключевого слова **template**, за которым следует список формальных типов параметров функции, заключенный в угловые скобки (<) и (>). Каждый формальный тип параметра предваряется ключевым словом **class**. Формальные типы параметров — это встроенные типы или типы, определяемые пользователем. Они используются для задания типов аргументов функции, для задания типов возвращаемого значения функции и для объявления переменных внутри тела описания функции. После шаблона следует обычное описание функции.

Приведем пример шаблона функции, возвращающей минимальный из трех передаваемых в нее параметров любого (но одинакового) типа:

```
template <class T>
T min(T x1, T x2, T x3)
{
    T lmin = x1;
    if (x2 < lmin)
        lmin = x2;
    if (x3 < lmin)
        lmin = x3;
    return lmin;
}
```

В заголовке шаблона этой функции объявляет единственный формальный параметр **T** как тип данных, который должен проверяться функцией **min**. В следующем далее заголовке функции этот параметр **T** использован для задания типа возвращаемого значения (**T min**) и для задания типов всех трех параметров **x1** - **x3**. В теле функции этот же параметр **T** использован для указания типа локальной переменной **lmin**.

Объявленный таким образом шаблон можно использовать, например, следующим образом:

```
int i1 = 1, i2 = 3, i3 = 2;
double r1 = 2.5, r2 = 1.7, r3 = 3.4;
AnsiString s1 = "строка 1", s2 = "строка 2", s3 = "строка 3";
Label1->Caption = min(i1,i2,i3);
Label2->Caption = min(r1,r2,r3);
Label3->Caption = min(s3, s2, s1);
```

Когда компилятор обнаруживает вызов **min** в исходном коде программы, этот тип данных, переданных в **min**, подставляется всюду вместо **T** в определении шаблона и C++ создает законченную функцию для определения максимального из трех значений указанного типа данных. Затем эта созданная функция компилируется. Таким образом, шаблоны играют роль средств генерации кода.

Например, при вызове функции с тремя целыми параметрами компилятор сгенерирует функцию:

```
int min(int x1, int x2, int x3)
{
    int lmin = x1;
    if (x2 < lmin)
        lmin = x2;
    if (x3 < lmin)
        lmin = x3;
    return lmin;
}
```

Приведенный шаблон будет работать для любых предопределенных или введенных пользователем типов, для которых определена операция отношения <.

Предупреждение

Каждый формальный параметр в определении шаблона должен хотя бы однажды появиться в списке параметров функции. Каждое имя формального параметра в списке определения шаблона должно быть уникальным. Отсутствие ключевого слова **class** перед каждым формальным параметром шаблона функции является ошибкой.

12.6 Области видимости переменных и функций

12.6.1 Правила, определяющие область видимости

Область видимости или *область действия* переменной или функции — это часть программы, в которой на нее можно ссылаться. Например, когда мы объявляем локальную переменную в блоке (определение блока см. в разделе 12.4.2), на нее можно ссылаться только в этом блоке или в блоке, вложенном в этот блок. Существуют четыре области действия идентификатора — *область действия функции*, *область действия файл*, *область действия блок* и *область действия прототип функции*.

Идентификатор, объявленный вне любой функции (на внешнем уровне), имеет *область действия файл*. Такой идентификатор «известен» всем функциям от точки его объявления до конца файла. Глобальные переменные, описания функций и

прототипы функций, находящиеся вне функции — все они имеют область действия файл.

Метки (идентификаторы с последующим двоеточием, например, **start:**) — единственные идентификаторы, имеющие *область действия функцию*. Метки можно использовать всюду в функции, в которой они появились, но на них нельзя ссылаться вне тела функции. Метки используются в структурах **switch** (как метки **case**) и в операторах **goto** (см. разделы 12.8.1.2 и 12.8.1.3). Метки относятся к тем деталям реализации, которые функции «прячут» друг от друга. Это скрывание — один из наиболее фундаментальных принципов разработки хорошего программного обеспечения.

Идентификаторы, объявленные внутри блока (на внутреннем уровне), имеют *область действия блок*. Область действия блок начинается объявлением идентификатора и заканчивается конечной правой фигурной скобкой блока. Если имеются вложенные блоки, то переменная внешнего блока видна и во вложенных блоках.

Локальные переменные, объявленные в начале функции, имеют область действия блок так же, как и параметры функции, являющиеся локальными переменными.

Любой блок может содержать объявления переменных. Если блоки вложены и идентификатор во внешнем блоке или идентификатор глобальной переменной идентичен идентификатору во внутреннем блоке, одноименный идентификатор внешнего блока или глобальный «невидим» (скрыт) до момента завершения работы внутреннего блока. Это означает, что пока выполняется внутренний блок, он видит значение своих собственных локальных идентификаторов, а не значения идентификаторов с идентичными именами в охватывающем блоке. Локальные переменные, объявленные как **static**, имеют область действия блок, несмотря на то, что они существуют с самого начала выполнения программы.

Из внутреннего блока можно получить доступ к одноименной глобальной переменной с помощью унарной операции разрешения области действия «**::**». Например, выражение **::I** означает глобальную переменную **I**, даже если в данном блоке объявлена локальная переменная **I**.

Единственными идентификаторами с *областью действия прототип функции* являются те, которые используются в списке параметров прототипа функции (см. раздел 12.5.1). Прототипы функций не требуют имен в списке параметров — требуются только типы. Если в списке параметров прототипа функции используется имя, компилятор это имя игнорирует. Идентификаторы, используемые в прототипе функции, можно повторно использовать где угодно в программе, не опасаясь двусмысленности.

При необходимости обеспечить видимость переменных, объявленных в одном модуле, из других модулей, в эти модули должны быть добавлены объявления соответствующих переменных (без их инициализации) со спецификацией **extern**. Для видимости из других модулей функций, объявленных в каком-то модуле, надо повторить в соответствующих модулях объявления этих функций. Это не относится к функциям, объявленным со спецификацией **static**. Такие функции невидимы в других модулях.

Если в модуле, описывающем функцию, ее объявление записано в заголовочном файле, то в другом модуле можно получить к ней доступ так, как описано выше, а можно и проще — включить директивой **#include** (см. раздел 12.2.1) этот заголовочный файл.

Полный список правил, определяющих видимость переменных и функций, вы можете найти в главе 1 в разделе 1.5.5.4.

12.6.2 Явное определение доступа с помощью объявлений `namespace` и `using`

Изложенные в предыдущем разделе правила определяют автоматически устанавливаемые области видимости. Однако такого неявного задания областей видимости иногда может быть недостаточно. Если речь идет о большом проекте, который создается несколькими разработчиками, всегда возможно перекрытие идентификаторов, определенных в разных местах программы. Поэтому желателен инструмент, позволяющий явным образом указывать области видимости идентификаторов.

Таким инструментом является объявление области видимости имен ключевым словом **namespace** и последующее объявление использования функций и переменных из той или иной области ключевым словом **using**.

Синтаксис объявления области видимости:

```
namespace имя_области
{
    объявления типов, переменных и функций
}
```

Например:

```
namespace A{
    int i = 1;
    void F1(int i)
    {
        Form1->Label1->Caption = "Область A: i = " + IntToStr(i);
    }
}
namespace B{
    int i = 2;
    void F1(int i)
    {
        Form1->Label1->Caption = "Область B: i = " + IntToStr(i);
    }
}
```

Приведенные операторы объявляют две области видимости с именами **A** и **B**. В обеих областях объявлены переменные **i** и функции **F1**.

Объявление области с тем же именем может повториться в программе и содержать объявления каких-то новых переменных и функций. Соответствующие идентификаторы добавятся в указанную область.

Доступ к объявленным переменным и функциям из любой точки файла может осуществляться несколькими способами. Самый простой — с помощью операции разрешения области действия (::). Например, оператор

```
B::F1(A::i);
```

вызовет функцию **F1** из области **B** и передаст в нее значение переменной **i** из области **A**.

Подобный доступ гибкий, но он требует каждый раз указывать область видимости. Если явное указание областей видимости сделано для того, чтобы устранить появившиеся в программе случайные наложения идентификаторов, то явное указание при каждом применении идентификатора соответствующей области действия потребует исправлений во многих местах программы и может привести к появлению ошибок. Более простой способ указания области действия — применение ключевого слова **using**. Одна из возможных форм применения **using**:

```
using namespace имя_области;
```

Например, если поместить в тексте оператор

```
using namespace A;
```

то все последующие операторы будут брать идентификаторы из области **A**. Тогда, например, размещенный где-то в тексте после `using` оператор

```
F1(i);
```

вызовет функцию **F1** из области **A** и передаст в нее значение переменной **i** из области **A**.

Операторы `using` могут иметь и другую форму, определяющую область для конкретного идентификатора:

```
using имя_области :: идентификатор;
```

Например, после операторов

```
using A::F1;  
using B::i;
```

оператор

```
F1(i);
```

вызовет функцию **F1** из области **A** и передаст в нее значение переменной **i** из области **B**.

При объявлении области видимости с помощью `namespace` в теле объявления могут присутствовать не только объявления переменных и функций, но и операторы `namespace`, определяющие некоторые внутренние области видимости, и операторы `using namespace`, ссылающиеся на ранее определенные области. Таким образом, области видимости могут быть вложенные. Например, объявления могут иметь вид:

```
namespace A {  
    ...  
}  
namespace B {  
    using namespace A;  
    ...  
    namespace C {  
        ...  
    }  
}
```

Здесь область **B** использует ранее объявленную область **A** и содержит внутри себя вложенную область **C**. Доступ к вложенным областям осуществляется последовательным применением операции разрешения области действия. Например:

```
using namespace B :: C;
```

12.7 Операции

12.7.1 Общее описание

Операции подобны встроенным функциям языка. Они применяются к выражениям — *операндам*. Большинство операций имеют два операнда, один из которых помещается перед знаком операции, а другой — после. Например, операция сложения «+» имеет два операнда: **X + Y** и складывает их. Такие операции называются бинарными. Существуют и унарные операции, имеющие только один операнд, помещаемый после знака операции. Например, запись **-X** означает применение к операнду **X** операции унарного минуса «in -».

В сложных выражениях последовательность выполнения операций определяется скобками, старшинством операций, а при одинаковом старшинстве — ассоциативностью операций. Эти вопросы будут обсуждены в разделе 12.7.15.

12.7.2 Арифметические операции

Арифметические операции применяются к действительным числам, целым числам и указателям. Определены следующие бинарные арифметические операции:

Обозначение	Операция	Типы операндов и результата	Пример
+	сложение	арифметический, указатель	$X + Y$
-	вычитание	арифметический, указатель	$X - Y$
*	умножение	арифметический	$X * Y$
/	деление	арифметический	X / Y
%	Остаток целочисленного деления	целый	$I \% 6$

Определены следующие унарные арифметические операции:

Обозначение	Операция	Типы операндов и результата	Пример
+	Унарный плюс (подтверждение знака)	арифметический	+7
-	Унарный минус (изменение знака)	арифметический	-X
++	инкремент	арифметический, указатель	i++; ++i
--	декремент	арифметический, указатель	i--; --i

Для арифметических операций действуют следующие правила.

Бинарные операции сложения (+) и вычитания (-) применимы к целым и действительным числам, а также к указателям.

В операции сложения указателем может быть только один из двух операндов. В этом случае второй операнд должен быть целым числом. Указатель, участвующий в операции сложения, должен быть указателем на элемент массива. В этом случае добавление к указателю целого числа эквивалентно сдвигу указателя на заданное число элементов массива.

В операции вычитания указатель на элемент массива может быть первым операндом (тогда второй операнд — целое число) или оба операнда могут быть указателями на элементы одного массива. Вычитание из указателя целого числа эквивалентно сдвигу указателя на заданное число элементов массива. Вычитание двух указателей возвращает число элементов массива, расположенных между теми элементами, на которые указывают указатели. Подробнее об арифметике указателей см. в главе 13 в разделе 13.7.

В операциях умножения (*) и деления (/) операнды могут быть любых арифметических типов. При разных типах операндов применяются стандартные правила автоматического приведения типов (см. раздел 13.2 главы 13). В операции вычисления остатка от деления (%) оба операнда должны быть целыми числами.

В операциях деления и вычисления остатка второй операнд не может быть равен нулю. Если оба операнда в этих операциях целые, а результат деления является не целым числом, то знак результата вычисления остатка совпадает со знаком первого операнда, а для операции деления используются следующие правила:

- 1. Если первый и второй операнд имеют одинаковые знаки, то результат операции деления — наибольшее целое, меньшее истинного результата деления.
- 2. Если первый и второй операнд имеют разные знаки, то результат операции деления — наименьшее целое, большее истинного результата деления.

Округление всегда осуществляется по направлению к нулю.

Унарные операции инкремента (++) и декремента (--) сводятся к увеличению (++) или уменьшению (--) операнда на единицу. Операции применимы к операндам, представляющим собой выражения любых арифметических типов или типа указателя. Причем выражение должно быть модифицируемым L-значением, т.е. должно допускать изменение. Например, ошибочным является выражение ++(a + b), поскольку (a + b) не является переменной, которую можно модифицировать.

Операции инкремента и декремента выполняются быстрее, чем обычное сложение и вычитание. Поэтому, если переменная a должна быть увеличена на 1, лучше применить операцию (++), чем выражения a = a + 1 или оператор a += 1, использующий описанную в разделе 12.7.3 операцию (+=).

Если операция инкремента или декремента помещена перед переменной, говорят о *префиксной форме записи* инкремента или декремента. Если операция инкремента или декремента записана после переменной, то говорят о *постфиксной форме записи*. При префиксной форме переменная сначала увеличивается или уменьшается на единицу, а затем это ее новое значение используется в том выражении, в котором она встретилась. При постфиксной форме в выражении используется текущее значение переменной, и только после этого ее значение увеличивается или уменьшается на единицу.

Например, в результате выполнения операторов

```
int i = 1, j;  
j = i++ * i++;
```

значение переменной i будет равно 3, а переменной j - 1. Оператор, присваивающий значение переменной j будет работать следующим образом: сначала значение i, равное 1, умножится само на себя, т.е. вычислится значение выражения в правой части оператора; затем это значение присвоится переменной j, а значение i увеличится на 1 в результате первой операции инкремента и еще раз увеличится на 1 в результате второй операции инкремента.

Если изменить эти операторы следующим образом:

```
int i = 1, j;  
j = ++i * ++i;
```

то результат будет другим: значение i будет равно 3, а значение j - 9. В этом случае оператор, присваивающий значение переменной j будет работать следующим образом: сначала выполнится первая операция инкремента и значение i станет равно 2; затем выполнится вторая операция инкремента и значение i станет равно 3; а затем это значение i умножится само на себя, т.е. вычислится значение выражения в правой части оператора и это значение присвоится переменной j.

12.7.3 Операции присваивания, отличие присваивания от метода Assign

В C++ определен ряд операций присваивания.

Обозначение	Операция	Типы операндов и результата	Пример
=	присваивание	любые	X = Y
+=	Присваивание со сложением	арифметические, указатели, структуры, объединения	X += Y

Обозначение	Операция	Типы операндов и результата	Пример
<code>-=</code>	Присваивание с вычитанием	арифметические, указатели, структуры, объединения	<code>X -= Y</code>
<code>*=</code>	Присваивание с умножением	арифметические	<code>X *= Y</code>
<code>/=</code>	Присваивание с делением	арифметические	<code>X /= Y</code>
<code>%=</code>	присваивание остатка целочисленного деления	целые	<code>X %= Y</code>
<code><=</code>	Присваивание со сдвигом влево	целые	<code>X <= Y</code>
<code>>=</code>	Присваивание со сдвигом вправо	целые	<code>X >= Y</code>
<code>&=</code>	присваивание с поразрядной операцией И	целые	<code>X &= Y</code>
<code>^=</code>	присваивание с поразрядной операцией исключающее ИЛИ	целые	<code>X ^= Y</code>
<code> =</code>	присваивание с поразрядной операцией ИЛИ	целые	<code>X = Y</code>

Помимо простой операции присваивания (`=`) все прочие являются составными операциями. Они присваивают первому операнду результат применения соответствующей простой операции, указанной перед символом «`=`», к первому и второму операндам.

Например, выражение `X += Y` эквивалентно выражению `X = X + Y`, но записывается компактнее и может выполняться быстрее. Аналогично определяются и другие операции присваивания: `X %= Y` эквивалентно `X = X % Y` и т.д. (см. соответствующие простые операции в разделах 12.7.2 и 12.7.6).

При записи составных операций присваивания между символом операции и знаком равенства пробел не допускается.

В операциях присваивания первый операнд не может быть нулевым указателем.

Операции присваивания возвращают как результат присвоенное значение. Благодаря этому они допускают сцепление. Например, вы можете написать:

```
A = (B = C = 1) + 1;
```

Выполняются операции присваивания справа налево. Поэтому приведенное выражение задаст переменным `B` и `C` значения 1, а переменной `A` - 2. Вычисляться это будет следующим образом. Сначала выполняются операции, заключенные в скобки, а из них первой — самая правая (т.е. `C = 1`). Эта операция вернет 1, так что далее будет выполнена операция `B = 1`. Она вернет значение 1, после чего выполнится операция сложения `1 + 1`. Полученное в результате значение 2 присвоится переменной `A`.

Применительно к указателям на объекты надо четко представлять различие между оператором присваивания и методом копирования `Assign` (см. соответствующий раздел в главе 16), свойственным многим классам объектов. Метод `Assign` используется следующим образом:

```
объект_приемник->Assign(объект_источник);
```

Например:

```
A->Assign(B);
```

Этот оператор копирует содержание объекта **B** (все его свойства) в объект **A**. Для тех же самых объектов **A** и **B** можно записать оператор присваивания:

```
A = B;
```

Различие между двумя приведенными операторами следующее. Метод **Assign** копирует содержимое одного объекта в другой. Таким образом в памяти будет иметься два объекта **A** и **B** одинакового содержания. А оператор присваивания, примененный к указателям (имя объекта — это указатель на объект), присваивает указателю **A** значение указателя **B**. Таким образом, и **A**, и **B** будут указывать на один и тот же объект в памяти. А тот объект, на который до выполнения этого оператора указывал **A**, может быть вообще потерян, если в программе где-то не хранится другой указатель на него.

12.7.4 Операции отношения и эквивалентности

Операции отношения и эквивалентности используются при сравнении двух операндов. Они возвращают **true** — истина, если указанное соотношение операндов выполняется, и **false** (0) — ложь, если соотношение не выполняется. Определены следующие операции отношения:

Обозначение	Операция	Типы операндов	Пример
==	Равно	арифметический, указатели	I == Max
!=	Не равно	арифметический, указатели	X != Y
<	Меньше чем	арифметический, указатели	X < Y
>	Больше чем	арифметический, указатели	Len > 0
<=	Меньше или равно	арифметический, указатели	Cnt <= I
>=	Больше или равно	арифметический, указатели	I >= 1

Операнды должны иметь совместимые типы, за исключением целых и действительных типов, которые могут сравниваться друг с другом.

Применять операции **<**, **<=**, **>**, **>=** к указателям имеет смысл, только если оба операнда указывают на элементы одного массива.

Операции **==** и **!=** могут применяться к указателям на любые объекты. В этом случае они вернут соответственно **true** и **false**, только если указатели указывают на один и тот же объект.

Следует предостеречь от довольно распространенной ошибки: случайного применения вместо операции эквивалентности (**==**) операции присваивания (**=**). Например, если вы по ошибке вместо оператора

```
if (A == 2) ...;
```

написали оператор

```
if (A = 2) ...;
```

то это не будет расценено как синтаксическая ошибка. Дело в том, что в C++ любое выражение, имеющее некоторое значение, может использоваться в условных операторах, в частности, в **if**. Если значение выражения 0, то оно трактуется как **false**. Любое другое значение трактуется как **true**. Поэтому результат операции **A = 2** будет трактоваться как **true** и независимо того, чему было равно значения **A** до выполнения этого ошибочного оператора, условие в операторе **if** всегда будет считать-

ся выполненным. К тому же эта ошибка приведет к несанкционированному изменению значения **A**.

К счастью, компилятор C++Builder замечает подобные недоразумения и при записи в операторе **if** операции присваивания на всякий случай делает замечание: «Possibly incorrect assignment» (Возможно некорректное присваивание). Это не ошибка, а только замечание. Так что если вы не обратите внимание на него, то потратите потом много времени на поиск ошибки в программе.

Хороший стиль программирования

Не пропускайте ни одного замечания компилятора, не проанализировав текст и не найдя причины, вызвавшей замечание. Это один из залогов построения надежного программного обеспечения.

12.7.5 Логические операции

Логические операции принимают в качестве операндов выражения скалярных типов и возвращают результат булева типа: **true** или **false** (0).

Обозначение	Операция	Пример
!	Отрицание	!A
&&	Логическое И	A && B
	Логическое ИЛИ	A B

Унарная операция логического отрицания (!) возвращает **true**, если операнд возвращает ненулевое значение. Таким образом, выражение **!A** эквивалентно выражению **A == 0**.

Операция логического И (&&) возвращает **true**, если оба ее операнда возвращают ненулевые значения. Если хотя бы один операнд возвращает 0 (**false**), то операция И также возвращает **false**. Поэтому для сокращения времени расчета, если первый операнд возвращает нуль, то второй операнд даже не вычисляется.

Операция логического ИЛИ (||) возвращает **true**, если хотя бы один ее операнд возвращает ненулевое значение. Если оба операнда возвращают 0 (**false**), то операция ИЛИ также возвращает **false**. Для сокращения времени расчета, если первый операнд возвращает ненулевое значение, то второй операнд даже не вычисляется.

12.7.6 Поразрядные логические операции

Поразрядные логические операции работают с целыми числами и оперируют с их двоичными представлениями, т.е. работают с двоичными разрядами операндов.

Обозначение	Операция	Пример
~	поразрядное отрицание	~ X
&	поразрядное И	X & Y
	поразрядное ИЛИ	X Y
^	поразрядное исключающее ИЛИ	X ^ Y
<<	поразрядный сдвиг влево	X << 2
>>	поразрядный сдвиг вправо	Y >> I

Операция поразрядного отрицания (~) инвертирует каждый бит операнда.

Поразрядные операции &, | и ^ работают в соответствии со следующей таблицей, где **E1** и **E2** — сравниваемые биты операндов:

E1	E2	E1 & E2	E1 ^ E2	E1 E2
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Операция поразрядного сдвига вправо (>>) сдвигает биты левого операнда на число разрядов, указанное правым операндом. При этом правые биты теряются. Если левый операнд представляет собой целое без знака, то левые освободившиеся биты заполняются нулями. В противном случае они заполняются символом знака. Сдвиг целого числа на **n** разрядов вправо эквивалентен целочисленному делению его на 2^n .

Операция поразрядного сдвига влево (<<) сдвигает биты левого операнда на число разрядов, указанное правым операндом. При этом левые биты теряются, а правые заполняются нулями. Сдвиг целого числа на **n** разрядов влево эквивалентен умножению его на 2^n .

12.7.7 Операция запятая (последование)

Операция запятая (,), называемая операцией последования, соединяет два произвольных выражения, которые вычисляются слева направо. Сначала вычисляется выражение левого операнда. Тип его результата считается **void**. Затем вычисляется выражение правого операнда. Значение и тип результата операции последования считается равным значению и типу правого операнда.

Например, фрагмент текста

```
a = 4;
b = a + 5;
```

можно записать как

```
a = 4, b = a + 5;
```

Можно рекурсивно соединить операциями запятая последовательность выражений:

```
выражение_1, выражение_2, ..., выражение_n
```

Выражения будут вычисляться слева направо, а самое правое выражение определит значение и тип всей этой последовательности.

Соединяться запятыми могут не только выражения присваивания, но и другие. Например, вызов функции с тремя параметрами может иметь вид

```
func(i, (j = 1, j + 4), k);
```

Здесь в качестве второго параметра передается значение операции последования, заключенной в скобки. В результате вызов производится со следующими аргументами: (**i**, **5**, **k**).

Операция последования используется в основном в операторах цикла **for** (см. раздел 12.8.2.1) для задания в заголовке некоторой совокупности действий. Например, цикл подсчета суммы элементов некоторого массива можно осуществить циклом **for** без использования операции последования:

```
int A[10], sum, i;
...
sum = A[0];
for (int i = 1; i < 10; i++)
    sum += A[i];
```

То же самое можно реализовать более компактно с помощью операции последования:

```
int A[10], sum, i;
...
for (i = 1, sum = A[0]; i < 10; sum += A[i], i++);
```

Здесь операция последования использована дважды: при задании действий, выполняемых перед началом цикла (задание начальных значений **i** и **sum**), и при описании действий, выполняемых в теле цикла (суммирование значений элементов в **sum** и инкремент счетчика **i**).

Хороший стиль программирования

Не используйте без нужды операцию последования (.). Применение ее оправдано только при объединении одинаковых по смыслу выражений в основном в операторах циклов. Более широкое применение операции последования ухудшает читаемость кода, маскирует ошибки, усложняет сопровождение программы.

12.7.8 Условная операция (?:)

Условная операция (?:) — единственная трехчленная (тернарная) операция в C++, имеющая три операнда. Ее синтаксис:

```
условие ? выражение_1 : выражение_2
```

Первый операнд является условием, второй операнд содержит значение условного выражения в случае, если условие истинно (возвращает ненулевое значение), а третий операнд равен значению условного выражения, если условие ложно (возвращает нуль). Например, оператор

```
Label1->Caption =
    grade > 3 ? "Вы хорошо знаете материал" : "Плохо";
```

в зависимости от значения переменной **grade** выдаст текст «Вы хорошо знаете материал» при значении **grade**, превышающем 3, и текст «Плохо» при меньшем значении **grade**.

Оператор с условной операцией выполняет фактически те же функции, что и оператор **if...else** (см. раздел 12.8.1.1). Но в ряде случаев применение условной операции компактнее и нагляднее оператора **if...else**. К тому же иногда условная операция может использоваться в таких ситуациях, когда применение оператора **if...else** синтаксически невозможно.

В условной операции условие может быть любым скалярным выражением. Условные выражения могут быть практически любого типа (арифметические, указатели, структуры, объединения), но типы двух выражений в операции должны быть согласованными. В качестве условных выражений могут также фигурировать какие-то исполняемые действия.

12.7.9 Операция sizeof

Операция **sizeof** определяет размер в байтах своего операнда — переменной, объекта, типа. Возвращаемый результат имеет тип **size_t (unsigned)**.

Операция имеет две формы:

```
sizeof выражение  
sizeof (имя_типа)
```

Например:

```
sizeof *Label1;  
sizeof(TLabel);  
sizeof a;  
sizeof (int);
```

Во всех случаях операция возвращает целое, равное числу байтов в объекте (***Label1**), типе (**TLabel**, **int**), переменной (**a**).

Надо учесть, что размер переменной, объекта, типа может изменяться в зависимости от машины и от используемой версии программного обеспечения. Поэтому во всех случаях, когда вам требуется знать размер объекта или типа, нельзя полагаться на документацию, а надо использовать операцию **sizeof**.

Если операндом является выражение, то **sizeof** возвращает суммарный объем памяти, занимаемый всеми переменными и константами, входящими в него. Если операндом является массив, то возвращается объем памяти, занимаемый всеми элементами массива (т.е. имя массива не воспринимается в данном случае как указатель). Число элементов в массиве можно определить выражением **sizeof array/sizeof array[0]**.

Если операндом является параметр, объявленный как тип массива или функции, то возвращается размер только указателя. К функциям операция **sizeof** не применима.

Если операция **sizeof** применяется к структуре или объединению, она возвращает общий объем памяти, включая все наполнение этого объекта.

12.7.10 Операция typeid

Операция **typeid** возвращает информацию времени выполнения **type_info**, о типе или выражении. Операция имеет две формы:

```
typeid( выражение )  
typeid( тип )
```

Если операндом является разыменованный указатель или ссылка на полиморфный тип, операция **typeid** возвращает динамический тип того реального объекта, на который ссылается указатель или ссылка. Если оператор не полиморфный, возвращается статический тип объекта.

12.7.11 Операции адресации (&) и косвенной адресации (*)

При работе с указателями и при передаче в функции параметров по ссылке используются операции (&) — адресации, и (*) — косвенной адресации или разыменования. Применение этих операций при работе с указателями подробно рассмотрено в разделе 13.7 главы 13. Применение их при передаче параметров в функции рассмотрено в разделе 12.5.2.

12.7.12 Операции разрешения области действия (::)

Операции разрешения области действия обозначаются двумя двоеточиями, записываемыми без пробела (::). Имеется две различных операции:

унарная:

```
:: переменная
```

и бинарная:

```
класс :: элемент_класса
```


Унарная операция разрешения области действия позволяет получить доступ к глобальной переменной из блока, в котором объявлена локальная переменная с тем же именем. Например, выражение `::I` означает глобальную переменную `I`, даже если в данном блоке или в одном из обрамляющих блоков объявлена локальная переменная `I`. Подробнее об областях действия (видимости) см. в разделе 12.6.

Бинарная операция разрешения области действия позволяет сослаться на данные-элемент или функцию-элемент класса, даже если имеются одноименные переменные или функции, определенные вне класса или в нескольких классах. Она используется также при описании функции-элемента вне класса. Вы можете увидеть автоматическое применение этой операции в любом модуле, создаваемом `C++Builder`, если взглянете на заголовок любого обработчика событий. Подробнее о применении бинарной операции разрешения области действия см. в главе 1 в разделе 1.5.5.3.

12.7.13 Операции доступа к элементам: точка (.) и стрелка (->)

Доступ к элементам структур и классов может осуществляться двумя операциями: операцией точки (.) или операцией стрелки (->). Если доступ осуществляется через объект, то используется операция точка. Например, если объект с именем `A` имеет свойство `Prop` и метод `F()`, то доступ к ним дается выражениями:

```
A.Prop
A.F()
```

Если доступ осуществляется через указатель на объект, что чаще всего практикуется для доступа к компонентам в `C++Builder`, то используется операция стрелка. Например:

```
Label1->Caption
Label1->Hide()
```

Правда, и в случае, если вы имеете указатель на объект, вы можете использовать операцию точка, но тогда вы сначала должны разыменовать указатель:

```
(*Label1).Caption
```

Впрочем, вряд ли подобное усложнение записи целесообразно.

12.7.14 Операции поместить в поток (<<) и взять из потока (>>)

Операции поместить в поток (<<) и взять из потока (>>) предназначены для работы с потоками, как со стандартными потоками `cout` и `cin`, используемыми в основном в консольных приложениях, так и с файлами (см. главу 13 раздел 13.9.3.1). В приведенных ниже примерах мы будем ориентироваться на то, что создается файловый поток `outfile` для вывода данных и файловый поток `infile` для чтения данных. Для этого должны быть выполнены операторы

```
#include <fstream.h>
// создание потока outfile, связанного с файлом "Test.dat"
ofstream outfile("Test.dat");
if(!outfile)
{
    ShowMessage("Файл не удастся создать");
    return;
}

... // операторы поместить в поток

outfile.close(); // закрытие файла
// создание потока infile, связанного с файлом "Test.dat"
ifstream infile("Test.dat");
```

```
if(!infile)
{
    ShowMessage("Файл не удается открыть");
    return;
}

...    // операторы взять их потока

infile.close();    // закрытие файла
```

Пояснения этих операторов см. в главе 13 в разделе 13.9.3.1.

Вывод в потоки может быть выполнен с помощью операции поместить в поток, т.е. перегруженной операции <<. Операция << перегружена для вывода элементов данных встроенных типов, для вывода строк и вывода значений указателей. Она позволяет также с помощью манипуляторов потока осуществлять вывод целых чисел в десятичном, восьмеричном и шестнадцатеричном форматах, вывод значений с плавающей запятой с различной точностью, с указанием по выводу десятичной точки, в экспоненциальном формате или в формате с фиксированной точкой, вывод данных с выравниваем относительно какой-либо границы поля указанной ширины, вывод данных с полями, заполненными заданными символами, вывод буквами в верхнем регистре в экспоненциальном формате и при выводе шестнадцатеричных чисел.

Операция << помещает в поток, являющийся ее первым операндом, аргумент, являющийся ее вторым операндом. Размещение в потоке происходит в текстовом виде. Например, оператор

```
outfile << "Привет!";
```

поместит в файл текст «Привет!». Операторы

```
int i = 25;
outfile << i;
```

поместят в файл текст "25".

Операция << возвращает ссылку на объект своего первого операнда, т.е. на поток. Это позволяет использовать сцепленные операции поместить в поток, например, оператор

```
outfile << "2 * 2 = " << (2 * 2);
```

поместит в файл текст "2 * 2 = 4". Это произойдет потому, что левая операция << поместит текст "2 * 2 = " и вернет **outfile**, после чего правая операция << будет иметь вид

```
outfile << (2 * 2);
```

и добавит к тексту результат своего правого операнда.

Проверить работу этого и рассматриваемых далее операторов можно, например, на следующем тестовом приложении. Разместите на форме компонент **Мемо**, кнопку и в обработчик ее события **OnClick** вставьте операторы:

```
char sin[80];

ofstream outfile("Test.dat");
if(!outfile)
{
    ShowMessage("Файл не удастся создать");
    return;
}

// операторы записи в файл, например:
outfile << "2 * 2 = " << 2 * 2;
// закрытие файла
outfile.close();
```

```
// открытие файла как входного потока
ifstream infile("Test.dat");
if(!infile)
{
    ShowMessage("Файл не удается открыть");
    return;
}
Memol->Clear();
while(!infile.eof())
{
    infile.getline(s1,80);
    Memol->Lines->Add(AnsiString(s1));
}
// закрытие файла
infile.close();
```

Подробное пояснение этих операторов вы найдете в главе 13 в разделе 13.9.3.1. А смысл их сводится к тому, что создается файл «Test.dat», связанный с потоком **outfile**, затем в него заносится операциями << некоторый текст, после чего файл закрывается. Затем он опять открывается, связываясь с потоком **infile**, и строки из него считываются и переносятся в окно **Memol**.

Продолжим рассмотрение операции <<. Последовательное применение операций поместить в поток (сцепленных или задаваемых самостоятельными операторами) приводит к занесению текстов в одну строку, как в рассмотренном выше примере. Если требуется перейти на новую строку, то можно или ввести в текст символ конца строки '\n' или применить манипулятор потока **endl** (сокращение от **end line** — конец строки). Например, операторы

```
outfile << "2 * 2 :\n" << (2 * 2) << "\n";
```

и

```
outfile << "2 * 2 : " << endl << (2 * 2) << endl;
```

дадут один и тот же результат: первая строка будет содержать текст «2 * 2 :», вторая — «4», а курсор файла будет переведен на третью строку.

В предыдущих примерах выводились константы и константные выражения. При выводе переменных все работает точно так же. Например, операторы

```
int i = 25, j = 2;
outfile << i << " * " << j << " = " << (i * j) << endl;
```

и операторы

```
int i = 25, j = 2;
char s[80] = "25 * 2 = ";
outfile << s << (i * j) << endl;
```

выводят в файл один и тот же текст: «25 * 2 = 50».

Предыдущий пример показывает, что вывод строки типа **char *** осуществляется просто записью в качестве правого операнда указателя на эту строку. Однако, для строк типа **AnsiString** (см. соответствующий раздел в главе 16) операция << не перегружена. Поэтому при выводе таких строк надо использовать приведение ее к типу **char *** с помощью метода **c_str**:

```
AnsiString sa = "Это строка AnsiString";
outfile << sa.c_str() << endl;
```

При выводе могут использоваться и достаточно сложные выражения. В приведенном ниже примере предполагается наличие двух окон редактирования **Edit1** и **Edit2**, в которые пользователь вводит целые числа, а программа выводит результат их сравнения:

```
int i = StrToInt(Edit1->Text);
int j = StrToInt(Edit2->Text);
```

```
outfile << i << (i == j ? " " : " не ") << "равно " << j  
    << endl;
```

В зависимости от введенных чисел будет выведен текст «... равно ...» или «... не равно ...».

Обратите внимание на то, что условный оператор заключен в скобки. Это необходимо делать, поскольку операция поместить в поток имеет сравнительно высокий приоритет (см. раздел 12.7.15) и без скобок она применилась бы только к переменной *i*, что вызвало бы сообщение о синтаксической ошибке.

Операция << позволяет выводить и указатели. Например, вы можете написать оператор

```
outfile << Mem01 endl;
```

и он выведет текст типа "0063F610" — шестнадцатеричный адрес объекта **Mem01**. Особым приемом надо выводить при необходимости указатель на строку типа **char ***. Если записать в операции поместить в поток сам указатель, например, *s*, то выведется не указатель, а содержимое строки. Так перегружена операция << при выводе строк. Если же нужен именно адрес, то перед именем указателя надо поместить операцию приведения типа (**void ***). Например:

```
outfile << (void *)s << endl;
```

Рассмотренные выше примеры далеко не исчерпывают возможностей вывода с помощью операции <<. При выводе можно использовать немало манипуляторов потоков, позволяющих форматировать текст, выводимый операцией <<. Ранее был рассмотрен только один манипулятор потока — **endl**. Описание других манипуляторов приведено в главе 13 в разделе 13.9.3.2.

Теперь остановимся на операции взять из потока (>>). Эта операция извлекает данные из потока, заданного ее левым операндом, и заносит их в переменную, заданную правым операндом. Операция возвращает поток, указанный как ее левый операнд. Благодаря этому допускаются сцепленные операции взять из потока. Например, оператор

```
infile >> i >> j;
```

прочтет, начиная с текущей позиции файла, связанного с потоком **infile**, два целых числа в переменные *i* и *j*. Если в текущей позиции файла первому из чисел предшествуют пробельные символы или разделители, то они будут пропущены. За окончание числа операция примет первый отличный от цифры символ, в частности, пробельный. Поэтому, если эти два числа были ранее записаны в файл например, оператором

```
outfile << i << ' ' << j << endl;
```

то они прочтутся нормально. Но если они были записаны оператором

```
outfile << i << j << endl;
```

т.е. без пробела, то их цифры будут слиты вместе и это составное число прочтется как *i*, а при чтении *j* произойдет ошибка.

Операцией >> можно вводить из файла строки в переменные типа **char ***. Например, операторы:

```
char s[80];  
infile >> s;
```

осуществляют чтение из файла в строку *s*. Но при этом читается не вся строка, а только одна лексема — последовательность символов, заканчивающаяся пробельным или разделительным символом. Для чтения целой строки имеются другие способы, отличные от операции взять из потока и рассмотренные в главе 13 в разделе 13.9.3.1.

Если при выполнении операции взять из потока считывается символ конца потока, то операция возвращает 0. Этим можно воспользоваться, чтобы, например, читать все содержимое файла, разбитое на лексемы:

```
while(infile>>s1)
{
    ...
}
```

12.7.15 Приоритет и ассоциативность операций

В сложных выражениях, содержащих несколько операций, последовательность их выполнения определяется прежде всего приоритетом операций. Имеется 16 уровней приоритета, приведенных ниже в таблице. Некоторые из этих уровней содержат всего по одной операции. Наивысший уровень имеют операции, приведенные в первой строке таблицы, низший — в последней. Операции, указанные в одной строке, имеют одинаковый уровень старшинства.

Там, где в таблице встречаются дубликаты операций (например, дубликаты имеют операции сложения и вычитания), первая относится к унарной операции, а вторая — к бинарной.

Если в выражении встречаются записанные подряд операции одного уровня старшинства, то последовательность их выполнения определяется ассоциативностью, которая может быть слева направо или справа налево.

Все операции, перечисленные в таблице, были рассмотрены в предыдущих разделах, кроме операций **new** и **delete**, которые будут рассмотрены в разделе 12.9.

Операция	Ассоциативность
() [] -> ::	слева направо
! ~ + - ++ -- & * sizeof new delete	справа налево
.* -> *	слева направо
* / %	слева направо
+ -	слева направо
<< >>	слева направо
< <= > >=	слева направо
== !=	слева направо
&	слева направо
^	слева направо
	слева направо
&&	слева направо
	справа налево
?:	слева направо
= *= /= %= += -= &= ^= = <<= >>=	справа налево
,	слева направо

Например, выражение $a + b * c / d$ будет выполняться как $a + ((b * c) / d)$. Сначала выполнятся операции умножения и деления, имеющие более высокий приоритет, чем операция сложения. Поскольку ассоциативность операций умножения

и деления слева направо, то прежде всего будет выполнено умножения $b * c$, а затем результат разделится на c . В заключение результат этого деления прибавится к a .

Вы можете легко изменять последовательность действий, применяя скобки, которые имеют очень высокий приоритет.

12.7.16 Перегрузка операций

Все операции C++ могут быть перегружены, кроме операций точка ($.$), разыменование ($*$), разрешение области действия ($::$), условная ($?:$) и `sizeof`.

Операции `=`, `[]`, `()` и `->` могут быть перегружены только как нестатические функции-элементы. Они не могут быть перегружены для перечислимых типов.

Все остальные операции можно перегружать, чтобы применять их к каким-то новым типам объектов, вводимым пользователем. Кроме того, многие операции уже перегружены в C++. Например, арифметические операции применяются к разным типам данных — целым числам, действительным и т.д., именно в результате того, что они перегружены.

Операции перегружаются путем составления описания функции (с заголовком и телом), как это делается для любых функций, за исключением того, что в этом случае имя функции состоит из ключевого слова **operator**, после которого записывается перегружаемая операция. Например, имя функции **operator+** можно использовать для перегрузки операции сложения.

Чтобы использовать операцию над объектами классов, эта операция *должна* быть перегружена, но есть два исключения. Операция присваивания (`=`) может быть использована с каждым классом без явной перегрузки. По умолчанию операция присваивания сводится к побитовому копированию данных-элементов класса. Такое побитовое копирование опасно для классов с элементами, которые указывают на динамически выделенные области памяти; для таких классов следует явно перегружать операцию присваивания. Операция адресации (`&`) также может быть использована с объектами любых классов без перегрузки; она просто возвращает адрес объекта в памяти. Но операцию адресации можно также и перегружать.

Перегрузка не может изменять старшинство и ассоциативность операций. Нельзя также изменить число операндов операции. Например, унарную операцию можно перегрузить только как унарную.

Перегрузка больше всего подходит для математических классов. Они часто требуют перегрузки значительного набора операций, чтобы обеспечить согласованность со способами обработки этих математических классов в реальной жизни. Например, было бы странно перегружать только сложение класса комплексных чисел, потому что обычно с комплексными числами используются и другие арифметические операции.

Цель перегрузки операций состоит в том, чтобы обеспечить такие же краткие выражения для типов, определенных пользователем, какие C++ обеспечивает с помощью богатого набора операций для встроенных типов. Однако, перегрузка операций не выполняется автоматически; чтобы выполнить требуемые операции, программист должен написать функции, осуществляющие перегрузки операций.

Хороший стиль программирования

Перегружайте операции так, чтобы они выполняли над объектами вашего класса ту же функцию или близкие к ней функции, что и операции, выполняемые над объектами встроенных типов.

Ниже приведен упрощенный пример создания класса комплексных чисел **Complex**, в котором переопределены операции сложения, вычитания и присваивания. Дается описание не всех функций, поскольку очевидно, что сложение и вычитание — операции идентичные с точностью до знака.

```

class Complex {
public:
    double Re;                // действительная часть
    double Im;                // мнимая часть
    Complex(double = 0.0, double = 0.0); // конструктор
// Операции сложения
    Complex operator+(const Complex &) const; // бинарная
    Complex operator+() const;                // унарная
// Операции вычитания
    Complex operator-(const Complex &) const; // бинарная
    Complex operator-() const;                // унарная
    Complex &operator=(const Complex &);      // присваивание
};

// Конструктор
Complex::Complex(double R, double I)
{
    Re = R;
    Im = I;
}
// Перегруженная бинарная операция сложения
Complex Complex::operator+(const Complex &X) const
{
    Complex R;
    R.Re = Re + X.Re;
    R.Im = Im + X.Im;
    return R;
}
...
// Перегруженная унарная операция вычитания
Complex Complex::operator-() const
{
    Complex R;
    R.Re = -Re;
    R.Im = -Im;
    return R;
}
// Перегруженная операция присваивания
Complex & Complex::operator=(const Complex &R)
{
    Re = R.Re;
    Im = R.Im;
    return *this; // возможность сцепления
}

```

В этом классе вводится два открытых данных-элемента: **Re** — действительная часть комплексного числа, и **Im** — мнимая часть. Конструктор по умолчанию задает действительную и мнимую части равными 0.

Оператор

```
Complex operator+(const Complex &) const;
```

объявляет прототип бинарной операции сложения. На то, что это операция бинарная, указывает наличие параметра — правого операнда. Когда компилятор встретит в тексте операцию **A + B**, примененную к переменным типа **Complex**, он, незримо для пользователя, заменит ее выражением **A.operator+(B)**.

Оператор

```
Complex operator+() const;
```

объявляет прототип унарной операции сложения, поскольку список параметров пуст.

Прототипы операций вычитания и присваивания строятся аналогично.

В реализации функции бинарного сложения создается локальная переменная **R** типа **Complex**, в которой формируется возвращаемое значение. В процессе формирования к данным левого операнда производится обращение просто по именам **Re** и **Im**, а второй операнд является параметром, передаваемым в функцию по ссылке. В заключение значение сформированной переменной возвращается как результат функции.

В функции операции присваивания просто данные параметра пересылаются в поля **Re** и **Im**. Обратите внимание на последнюю строку, которая возвращает ***this**. Указатель **this** является указателем на объект данного класса. Подобный возврат ссылки необходим, чтобы можно было использовать сцепленные операции присваивания. Рассмотрим это подробнее.

Если компилятор встречает в тексте выражение **A = B**, примененное к переменным типа **Complex**, он заменяет его выражением **A.operator=(B)**. А что произойдет, если встретится выражение **A = B = C**? Поскольку ассоциативность операции присваивания справа налево, то сначала заменится вторая часть выражения на **B.operator=(C)**. После замены первого знака равенства получится выражение **A.operator=(B.operator=(C))**. Для того, чтобы это работало, нужно, чтобы выражение возвращало ссылку на объект **B**. Это и делается, возвращением в функции ссылки ***this**. Тогда обеспечивается правильное выполнение сцепленных присваиваний.

С описанным классом вы можете, например, выполнять такие действия:

```
Complex A(1,1), B(2,2), C,D;  
A = -A;  
C = A + B + B;  
D = A - B;  
A = B = C;
```

12.8 Операторы

12.8.1 Операторы передачи управления

12.8.1.1 Условные операторы выбора if

Оператор **if** предназначен для выполнения тех или иных действий в зависимости от истинности или ложности некоторого условия. Условие задается выражением, имеющим результат булева типа.

Оператор имеет две формы: **if** и **if...else**. Форма **if** имеет вид:

```
if (условие) оператор;
```

Скобки, обрамляющие условие, обязательны.

Условием может быть выражение, преобразуемое в булев тип. Если условие истинно (возвращает **true** - ненулевое значение), то указанный в конструкции **if** оператор выполняется. В противном случае управление сразу передается следующему за конструкцией **if** оператору. Например, в результате выполнения операторов

```
C = A;  
if (B > A) C = B;
```

переменная **C** станет равна максимальному из чисел **A** и **B**, поскольку оператор **C = B** будет выполнен только при **B > A**.

Поскольку в C++ арифметическое (целое или действительное) значение может преобразовываться к булеву (любое ненулевое значение воспринимается как **true**, а нулевое — как **false**), то условие может иметь целый тип. Например:

```
int a, b, c;  
...  
if(a - b/c) ...;
```

В данном случае условие **if(a - b/c)** эквивалентно **if(a == b/c)**, поскольку **a - b/c** возвращает нуль при равенстве **a** и **b/c**. Аналогичные условия формально можно записывать и для действительных чисел:

```
double a, b, c;  
...  
if(a - b/c) ...;
```

Но из-за ошибок округления это может не сработать, даже если теоретически значения **a** и **b/c** должны совпадать.

Хороший стиль программирования

В условных операторах выбора можно сравнивать целые числа и записывать целые выражения. Но никогда не записывайте действительные выражения. Из-за ошибок округления результаты их вычисления могут не быть равным нулю, даже когда чисто теоретически результат дает нуль.

В условии можно объявлять переменные. Например:

```
if (int v = func(a)) ...;
```

В этом случае область действия и существования объявленной переменной — только данная структура **if**, включая ее выполняемый оператор.

Форма конструкции **if...else** имеет вид:

```
if (условие) оператор;  
else оператор2;
```

Если условие возвращает **true**, то выполняется первый из указанных операторов, в противном случае выполняется второй оператор. Обратите внимание, что в конце первого оператора перед ключевым словом **else** ставится точка с запятой.

Приведем примеры.

```
if (J == 0)  
    ShowMessage("Деление на нуль");  
else  
    Result = I/J;
```

В качестве и первого, и второго оператора могут, конечно, использоваться и составные операторы:

```
if (J == 0)  
{  
    ShowMessage("Деление на нуль");  
    Result = 0;  
}  
else  
    Result = I/J;
```

Опять обратите внимание, что после фигурной скобки перед **else** точка с запятой не ставится.

При вложенных конструкциях **if** могут возникнуть неоднозначности в понимании того, к какой из вложенных конструкций **if** относится элемент **else**. Компилятор всегда считает, что **else** относится к последней из конструкций **if**, в которой не было раздела **else**.

Например, в конструкции

```
if (условие1)  
    if (условие2)  
        оператор1;  
    else оператор2;
```

else будет отнесено компилятором ко второй конструкции **if**, т.е. **оператор2** будет выполняться в случае, если первое условие истинно, а второе ложно. Иначе говоря, вся конструкция будет прочитана как

```
if (условие1)
{
    if (условие2) оператор1;
    else оператор2;
}
```

Если же вы хотите отнести **else** к первому **if**, это надо записать в явном виде с помощью фигурных скобок:

```
if (условие1)
{
    if (условие2) оператор1;
}
else оператор2;
```

12.8.1.2 Условный оператор множественного выбора **switch**

Оператор **switch** позволяет провести анализ значения некоторого выражения и в зависимости от его значения выполнить те или иные действия. В общем случае формат записи оператора **switch** следующий:

```
switch (выражение_выбора) {
    case значение_1 : оператор_1;
                        break;          // не обязательно
    ...
    case значение_n : оператор_n;
                        break;          // не обязательно
    default : оператор;                // не обязательно
}
```

В этой конструкции выражение выбора должно иметь порядковый тип — целый, перечислимый и т.д. Поэтому, например, нельзя использовать выражения, возвращающие действительные числа или строки.

Значения, указываемые в метках **case**, должны быть константными выражениями, соответствующими возможным значениям выражения выбора. После значения ставится двоеточие «:», а затем пишется оператор (может писаться составной оператор), который должен выполняться, если выражение приняло указанное в метке значение.

Если значение выражения выбора совпало со значением, указанным в одной из меток **case**, то выполняется оператор, записанный после этой метки, после чего, если не принять соответствующих мер, будут выполняться все последующие операторы остальных меток. Поскольку это обычно нежелательно, то, как правило, после оператора, который должен выполняться, записывают оператор

```
break;
```

Он прерывает выполнение структуры **switch** и управление передается следующему за ней оператору.

Если значение выражения выбора не соответствует ни одному из перечисленных в метках, то выполняется оператор, следующий за меткой **default**. Впрочем, метка **default** не обязательно должна включаться в структуру **switch**. В этом случае, если не нашлось соответствующего значения выражения выбора, то ни один оператор не будет выполнен.

Значения в метках могут содержать константы и константные выражения, которые совместимы по типу с объявленным выражением и которые компилятор может вычислить заранее, до выполнения программы. Недопустимо использование переменных и многих функций. В метках не допускается повторение одних и тех же значений, поскольку в этом случае выбор был бы неоднозначным.

Приведенный ниже пример анализирует переменную **Key** типа **char**, содержащую символ, введенный пользователем в ответ на некоторый вопрос. При положительном ответе вызывается процедура **FYes**, при отрицательном — **FNo**, при иных ответах отображается сообщение об ошибке.

```
switch (Key) {
  case 'y': case 'Y': { FYes(); break; }
  case 'n': case 'N': { FNo(); break; }
  default :   ShowMessage("Ошибочный ответ");
}
```

Обратите внимание, что при необходимости выполнять одинаковые действия при нескольких значениях выражения выбора, надо размещать подряд несколько меток **case**.

12.8.1.3 Оператор передачи управления **goto**

Оператор **goto** позволяет прервать обычный поток управления и передать управление в произвольную точку кода, помеченную специальной меткой. В свое время при появлении концепции структурного программирования на оператор **goto** обрушился поток критики и его применение стало рассматриваться как дурной тон. Действительно, чрезмерно широкое применение **goto** делает структуру программы крайне запутанной и затрудняет ее сопровождение. Однако, во многих случаях стремление обойтись без оператора **goto** не только не упрощает код, а еще более его запутывает. Так что этот оператор, безусловно, имеет право на существование.

Хороший стиль программирования

Без особой нужды старайтесь обходиться без применения оператора **goto**. Однако, не вводите это в принцип. В некоторых случаях применение **goto** не только не запутывает программу, но, наоборот, делает ее более понятной и облегчает отладку. Старайтесь размещать передачу управления и метки, на которые передается управление, недалеко друг от друга и как следует выделяйте их и снабжайте соответствующими комментариями.

Метка в тексте программы обозначается идентификатором с последующим двоеточием. Например,

```
Lbegin:
```

Метка отмечает точку, в которую передается управление оператором **goto**. Метка может располагаться в любом месте блока, как после оператора **goto**, передающего на нее управление, так и до этого оператора. Надо только иметь в виду, что передача управления извне внутрь цикла может приводить к непредсказуемым последствиям, так что таких ситуаций следует избегать.

Метки имеют область действия функцию. Метки можно использовать всюду в функции, в которой они появились, но на них нельзя ссылаться вне тела функции. Метки используются также в структурах **switch** (как метки **case** — см. раздел 12.8.1.2).

После метки следует оператор, на который передается управление.

Сам оператор **goto** имеет форму:

```
goto метка;
```

Таким образом, организация работы с операторами **goto** может выглядеть, например, так:

```
goto L1;
...
second: ...
...
L1: ...
...
```

```
if (...) goto L1;  
else goto second;
```

При этом, как видно, можно ссылаться на метки, расположенные после или до оператора **goto**.

12.8.2 Операторы циклов

12.8.2.1 Оператор **for**

Оператор **for** обеспечивает циклическое повторение некоторого оператора (в частности, составного оператора) заданное число раз. Повторяемый оператор называется *телом цикла*. Повторение цикла обычно определяется некоторой управляющей переменной (счетчиком), которая изменяется при каждом выполнении тела цикла. Повторение завершается, когда управляющая переменная достигает заданного значения.

Синтаксис структуры **for**:

```
for (выражение1; выражение2; выражение3) оператор;
```

где **выражение1** задает начальное значение переменной, управляющей циклом, **выражение2** является условием продолжения цикла, а **выражение3** изменяет управляющую переменную.

Структура **for** работает следующим образом. Сначала выполняется **выражение1** (оно может состоять и из ряда выражений, разделенных запятой т.е. может использоваться операция последования — см. раздел 12.7.7). Это выражение задает начальные значения переменной (или переменных) цикла. Затем проверяется **выражение2** — условие продолжения цикла. Если условие истинно (возвращает **true** — ненулевое значение), то выполняется тело цикла — оператор, записанный в структуре **for**. После завершения тела цикла выполняется **выражение3**, определяющее обычно изменение переменной цикла. Затем опять проверяется условие, записанное как **выражение2**, и при истинности этого условия выполнение цикла продолжается. Как только в каком-нибудь цикле **выражение2** вернет **false** (нулевое значение), цикл прерывается и управление передается оператору, расположенному следом за структурой **for**.

Приведем примеры использования цикла **for**. Следующие операторы вычисляют максимальное значение и сумму элементов, расположенных в массиве целых чисел **Data** размерностью 10:

```
int Max, Sum;  
Max = Sum = Data[0];  
for(int i = 1; i < 10; i++)  
{  
    if (Data[i] > Max) Max = Data[i];  
    Sum += Data[i];  
}
```

Здесь первое выражение в структуре **for** вводит целую переменную **i**, являющуюся счетчиком циклов, и инициализирует ее значением 1. Второе выражение проверяет условие завершения цикла. В данном случае цикл должен завершиться, когда переменная **i**, используемая в теле цикла как индекс массива, примет значение, большее 9. Третье выражение структуры **for** увеличивает после каждого выполнения цикла значение **i** на 1 с помощью операции инкремента.

В данном случае переменная **i** объявлена в заголовке структуры **for**. Значит ее область действия только эта структура. После завершения циклов переменная **i** удаляется из памяти.

При использовании компилятора **BCC32.EXE**, запускаемого из командной строки, подобное уничтожение локальной переменной, объявленной в цикле, можно отменить опцией **-Vd**. Но вряд ли это имеет смысл делать.

Теперь рассмотрим пример использования в структуре **for** операции запятая (см. раздел 12.7.7). Пусть в приведенном выше примере нам надо найти только сумму элементов массива. Тогда, если объявить переменные **i** и **Sum** до начала цикла, собственно цикл можно весь разместить в заголовке структуры **for**:

```
int Sum,i;
for(Sum = Data[0],i = 1; i < 10; Sum += Data[i++]);
```

В этом примере первое выражение структуры **for** включает в себя два оператора, разделенных операцией запятая и задающих начальные значения переменной **Sum**, накапливающей сумму, и переменной цикла **i**. Третье выражение структуры **for** объединяет в одном операторе формирование суммы и постфиксный инкремент переменной цикла **i**. После структуры **for** стоит точка с запятой, что означает пустое тело цикла.

В приведенных примерах переменная цикла увеличивалась на единицу при каждом цикле. Можно, конечно, организовывать циклы с уменьшением переменной. Ниже приведен такой пример. В нем приведена программа, которая берет строку, записанную в окне редактирования **Edit1**, шифрует ее сложением по операции исключающее ИЛИ каждого символа строки с произвольным ключом и возвращает строку в окно редактирования. Если повторно применить эту процедуру с тем же ключом к зашифрованной строке, то будет произведена дешифровка и в окне отобразится исходная строка.

```
AnsiString s;
char Key = 'A';
s = Edit1->Text;
for (int i = s.Length(); i > 0; s[i--] = s[i] ^ Key);
Edit1->Text = s;
```

В этом примере начальное значение переменной **i** задано равным числу символов в строке, полученному применением функции **Length()**. В дальнейшем при каждом выполнении цикла **i** уменьшается на 1 постфиксной операцией декремента. В этом случае целесообразно именно уменьшение счетчика цикла, поскольку в противном случае во втором выражении структуры **for** пришлось бы каждый раз проверять с помощью функции **Length()**, не кончилась ли строка. А в приведенном варианте обращение к **Length()** производится всего один раз.

Выражения в структуре **for** являются необязательными. Иногда может отсутствовать первое выражение, если начальное значение управляющей переменной задано где-то в другом месте программы. Если отсутствует второе выражение, предполагается, что условие продолжения цикла всегда истинно и таким образом создается бесконечно повторяющийся цикл. Выйти из такого цикла можно, проверив в теле цикла какие-то условия и прервав выполнение передачей управления за пределы цикла оператором **goto** или применить другие способы прерывания, рассмотренные в разделе 12.8.2.4. Может отсутствовать в структуре **for** и третье выражение, если приращение переменной осуществляется операторами в теле структуры или если приращение не требуется.

При пропуске какого-то из выражений, точка с запятой после пропущенного выражения (кроме третьего) должна писаться. Например, в заголовке

```
for(; i<10;) ...;
```

пропущено первое условие и третье.

Если условие продолжения цикла не удовлетворяется с самого начала, то операторы тела структуры **for** не выполняются ни разу.

Операторы цикла **for** могут быть вложенные. Следующий пример содержит три вложенных цикла **for**, осуществляющих вычисление матрицы **Mat**, равной произведению двух квадратных матриц **Mat1** и **Mat2** размером **M** на **M**. Все матрицы представлены двумерными массивами. Формула для вычисления:

$$Mat[I, J] = \sum_{K=1}^M Mat1[I, K] \cdot Mat2[K, J].$$

```
int I, J, K, X;
for(I = 1; I <= M; I++)
    for(J = 1; J <= M; J++)
    {
        X = 0;
        for(K = 1; K <= M; K++)
            X += Mat1[I][K] * Mat2[K][J];
        Mat[I][J] = X;
    }
```

12.8.2.2 Оператор do...while

Структура **do...while** используется для организации циклического выполнения оператора или совокупности операторов, называемых телом цикла, до тех пор, пока не окажется нарушенным некоторое условие. Синтаксис управляющей структуры **do...while**:

```
do оператор while (условие);
```

Структура работает следующим образом. Выполняется оператор тела цикла. Затем вычисляется **условие** — выражение, которое должно возвращать результат булева типа. Если выражение возвращает **true** (не нулевое значение), то повторяется выполнение тела цикла и после этого снова вычисляется выражение. Такое циклическое повторение цикла продолжается до тех пор, пока проверяемое выражение не вернет **false** (нуль). После этого цикл завершается и управление передается оператору, следующему за структурой **do...while**.

Поскольку проверка выражения осуществляется после выполнения тела цикла, то цикл будет заведомо выполнен хотя бы один раз, даже если выражение сразу ложно. С другой стороны, программист должен быть уверен, что выражение рано или поздно вернет **false**. Если этого не произойдет, то программа «заиклится», т.е. цикл будет выполняться бесконечно. Иногда такие бесконечные циклы используются. Но в этом случае внутри тела цикла должно быть предусмотрено его прерывание в какой-то момент, например, оператором **break** или другими способами, рассмотренными в разделе 12.8.2.4.

Обычно оператор **do** целесообразно использовать для организации поиска среди множества объектов такого, который обладает каким-то определенным свойством. Причем заранее должно быть известно, что множество объектов не пустое, т.е. хотя бы один объект в нем имеется. К тому же должен быть критерий, позволяющий проверить, не является ли текущий объект последним. Тогда тело цикла включает операторы перехода к новому объекту и какой-то его обработки, а условие **while** включает проверку, является ли объект не последним и отсутствуют ли у него искомые свойства. Если объект последний или искомые свойства найдены, выполнение цикла прерывается. Если же объект не последний и искомые свойства у него не найдены, осуществляется переход к следующему объекту.

Если множество проверяемых объектов может быть пустым, следует использовать другой оператор цикла — **while** (см. раздел 12.8.2.3). Если число повторений циклов заранее известно, лучше применять оператор **for** (см. раздел 12.8.2.1).

Ниже приведен пример, в котором в файле **File1.txt** ищется строка, содержащая фрагмент текста (последовательность символов с учетом регистра), указанный пользователем в окне редактирования **Edit1**. Проверка наличия в строке заданного фрагмента проверяется функцией **strstr**. Окончание файла проверяется функцией **feof**.

```
FILE *F;
char S[256] = "";
AnsiString SKey = Edit1->Text;
if ((F = fopen("File1.txt", "r")) == NULL)
```



```
{
    ShowMessage("Файл не найден");
    return;
}

do
    fgets(S,256,F);
while(!feof(F) && (strstr(S,SKey.c_str()) == NULL));

fclose(F);
if (strstr(S,SKey.c_str()) == NULL)
    ...
```

Цикл будет выполняться, до тех пор, пока не достигнут конец файла и пока функция **strstr** возвращает **NULL** (фрагмент не найден). Если хотя бы одно из этих условий нарушается (достигнут конец файла или найден фрагмент), выполнение цикла прекращается.

12.8.2.3 Оператор while

Оператор **while** используется для организации циклического выполнения тела цикла, пока выполняется некоторое условие. Синтаксис структуры **while**:

```
while (условие) оператор;
```

Структура работает следующим образом. Сначала вычисляется условие, которое должно возвращать результат булева типа. Если выражение возвращает **true** (ненулевое значение), то выполняется оператор тела цикла, после чего опять вычисляется выражение, определяющее условие. Такое циклическое повторение выполнения оператора и проверки условия продолжается до тех пор, пока условие не вернет **false** (нуль). После этого цикл завершается и управление передается оператору, следующему за структурой **while**.

Поскольку проверка выражения осуществляется перед выполнением оператора тела цикла, то, если условие сразу ложно, оператор не будет выполнен ни одного раза.

Программист должен быть уверен, что выражение рано или поздно вернет **false**. Если этого не произойдет, то программа «зациклится», т.е. цикл будет выполняться бесконечно. Иногда такие бесконечные циклы используются. Но в этом случае внутри тела цикла должно быть предусмотрено его прерывание в какой-то момент, например, оператором **break**, прерывающим цикл, или другими способами, рассмотренными в разделе 12.8.2.4.

Часто оператор **while** используется для организации поиска среди множества объектов такого, который обладает каким-то определенным свойством. Причем не исключается, что множество объектов может быть пустым, т.е. не содержащим ни одного объекта. К тому же должен быть критерий, позволяющий проверить, не является ли текущий объект последним. Тогда тело цикла включает операторы перехода к новому объекту и какой-то его обработки, а условие **while** включает проверку, является ли объект не последним и не обладает ли он искомым свойством. Если одно из этих условий нарушается (объект последний или имеет искомое свойство), выполнение цикла прерывается.

Ниже повторен приведенный в предыдущем разделе пример поиска в файле **File1.txt** фрагмента текста, указанного пользователем в окне редактирования **Edit1**. Но если в предыдущем разделе для организации цикла использовался оператор **do...while**, то в данном случае использован оператор **while**. Этот оператор здесь более уместен, поскольку проверка конца файла осуществляется до начала цикла, т.е. до чтения из него строки. Поэтому все будет нормально работать даже в случае, если файл окажется пустым и в нем не будет ни одной строки.

```

FILE *F;
char S[256] = "";
AnsiString SKey = Edit1->Text;
if((F = fopen("File1.txt", "r")) == NULL)
{
    ShowMessage("Файл не найден");
    return;
}

while(!feof(F) && (strstr(S, SKey.c_str()) == NULL))
    fgets(S, 256, F);

fclose(F);
if (strstr(S, SKey.c_str()) == NULL)
    ...

```

В данном случае можно использовать и цикл **for**:

```

for(; !feof(F) && (strstr(S, SKey.c_str()) == NULL);
    fgets(S, 256, F));

```

но цикл **while** выглядит наиболее естественным.

12.8.2.4 Прерывание цикла: операторы **break**, **Continue**, **return**, функция **Abort**

В некоторых случаях желательно прервать повторение цикла, проанализировав какие-то условия внутри него. Это может потребоваться в тех случаях, когда проверки условия окончания цикла громоздки, требуют многоэтапного сравнения и сопоставления каких-то данных и все эти проверки просто невозможно разместить в выражении условия операторов **for**, **do** или **while**.

Один из возможных вариантов решения этой задачи — ввести в код какой-то флаг окончания (переменную). При выполнении всех условий окончания этой переменной присваивается некоторое условное значение. Тогда условие в операторах **for**, **do** или **while** сводится к проверке, не равно ли значение этого флага принятому условному значению.

Другой способ решения задачи — использование оператора **break**. Он используется как в операторах цикла, так и в структурах **switch**. Оператор **break** прерывает выполнение тела любого цикла **for**, **do** или **while** и передает управление следующему за циклом выполняемому оператору.

Например, цикл в рассмотренном в предыдущих разделах примере поиска текста в файле мог бы быть организован следующим образом:

```

while(!feof(F))
{
    fgets(S, 256, F);
    if(strstr(S, SKey.c_str()) != NULL) break;
}

```

Еще один способ прерывания цикла — использование оператора **goto**, передающего управление какому-то оператору, расположенному вне тела цикла.

Для прерывания циклов, размещенных в функциях, можно воспользоваться оператором **return**. В отличие от оператора **break**, оператор **return** прервет не только выполнение цикла, но и выполнение той функции, в которой расположен цикл.

Прервать выполнение цикла, а заодно — и блока, в котором расположен цикл, можно также генерацией какого-то исключения (см. раздел 12.10). Наиболее часто в этих целях используется процедура **Abort**, генерирующая «молчаливое» исключение, не связанное с каким-то сообщением об ошибке.

Описанные способы прерывали выполнение цикла. Имеется еще процедура **Continue**, которая прерывает только выполнение текущей итерации, текущего выполнения тела цикла и передает управление на следующую итерацию.

Чтобы продемонстрировать применение **Continue**, усложним рассмотренный ранее пример поиска заданного фрагмента в текстовом файле. Пусть, например, мы хотим найти заданный фрагмент не в любой строке файла, а только в такой, которая начинается с символа «*». Тогда поиск можно было бы организовать следующим образом:

```
while(!feof(F))
{
    fgets(S,256,F);
    if(S[0] != '*') continue;
    if(strstr(S,SKey.c_str()) != NULL) break;
}
```

В этом варианте при первом символе в строке, отличном от «*», текущая итерация прерывается и поиск в такой строке не производится. Таким образом, не тратится время на выполнение функции **strstr** для строк, в которых искать фрагмент не нужно.

12.9 Динамическое распределение памяти

Динамическое распределение памяти широко используется для экономии вычислительных ресурсов. Те переменные или объекты, которые становятся ненужными, уничтожаются, а освобожденное место используется для новых переменных или объектов. Это особенно эффективно в задачах, в которых число необходимых объектов зависит от обрабатываемых данных или от действий пользователя, т.е. заранее не известно. В этих ситуациях остается только два выхода: заранее с запасом отвести место под множество объектов или использовать динамическое распределение памяти, создавая новые объекты по мере надобности. Первый путь, конечно, неудовлетворительный, поскольку связан с излишними затратами памяти и в то же время накладывает на размерность задачи необоснованные ограничения.

Для динамического распределения выделяется специальная область памяти — **heap**. Динамическое распределение памяти в этой области может производиться несколькими способами: с помощью библиотечных функций **malloc**, **calloc**, **realloc**, **free** или с помощью операций **new** и **delete**.

Указанные функции объявлены в файле **stdlib.h** или **alloc.h**. Объявление функции **malloc** следующее:

```
void *malloc(size_t size);
```

Функция выделяет в **heap** блок размером в **size** байтов. В случае успешного выделения памяти функция возвращает указатель на выделенный блок. Если не хватило места для блока требуемого размера или если **size = 0**, возвращается **NULL**.

Другая функция — **calloc** объявлена следующим образом:

```
void *calloc(size_t nitems, size_t size);
```

Функция выделяет память под **nitems** объектов, размер каждого из которых равен **size**. Таким образом общий объем выделяемой памяти составляет **nitems * size**. Выделенная память инициализируется нулями. В случае успешного выделения памяти функция возвращает указатель на выделенный блок. Если не хватило места для блока требуемого размера или если **size = 0** или **nitems = 0**, возвращается **NULL**.

Еще одна функция — **realloc** позволяет изменить размер ранее выделенного блока памяти. Функция объявлена следующим образом:

```
void *realloc(void *block, size_t size);
```

Она изменяет размер блока в `heap`, на который указывает **block**, до размера **size**. При этом предполагается, что **block** указывает блок памяти, выделенной ранее функциями **malloc**, **calloc** или **realloc**. Если же аргумент **block** задан равным **NULL**, то функция **realloc** работает так же, как описанная выше функция **malloc**.

Если размер **size** задан равным нулю, то выделенный ранее блок, на который указывает **block**, освобождается, а функция возвращает **NULL**. Таким образом, функция с **size** равным 0 может использоваться не для выделения памяти, а для освобождения памяти, выделенной ранее.

Если блок нового размера не может быть выделен, то функция **realloc** возвращает **NULL**. Если же память выделилась успешно, то возвращается адрес выделенного блока. При этом он может отличаться от начального значения **block**, поскольку функция при необходимости осуществляет копирование содержимого блока в новое место.

Функция **free** объявлена следующим образом:

```
void free(void *block);
```

Она освобождает блок памяти, выделенный ранее функциями **malloc**, **calloc** или **realloc**, на который указывает **block**.

Рассмотрим примеры использования описанных функций. Следующий код динамически выделяет функцией **malloc** память под строку, а затем, после выполнения с ней каких-то операций, освобождает выделенную память.

```
#include <stdio.h>
#include <alloc.h>
char *str;

// str - указатель на строку, под которую выделена память
str = (char *) malloc(100);
...
// освобождение памяти
free(str);
```

В этом примере можно было бы использовать для выделения памяти функцию **calloc**:

```
str = (char *) calloc(100, sizeof(char));
```

Размер выделенной функциями **malloc** или **calloc** памяти можно было бы изменить, например, следующим оператором:

```
str = (char *) realloc(str, 20);
```

Впрочем, к тому же результату привел бы и более простой оператор:

```
realloc(str, 20);
```

Необходимо помнить, что рассмотренные функции возвращают **NULL** (0), если память не удалось выделить. Поэтому прежде, чем использовать возвращенные ими указатели, надо обязательно проверять, не равны ли они **NULL**. Иначе возможны очень тяжелые ошибки при работе программы.

Теперь рассмотрим другой подход к динамическому распределению памяти: операции **new** и **delete**.

Операция **new** работает аналогично функции **malloc**, но лучше использовать именно ее, а не **malloc**. Это пожелание становится безусловной необходимостью, если речь идет о динамическом размещении в памяти объектов библиотеки компонентов C++Builder.

Операция **new** имеет следующий синтаксис:

```
<::> new <размещение> тип <(инициализатор)>
```

```
<::> new <размещение> (тип) <(инициализатор)>
```

Операция возвращает указатель на динамически размещенный в памяти объект.

Все элементы, заключенные в описании синтаксиса в угловые скобки, являются необязательными. Операция разрешения области действия (::) позволяет обратиться к глобальной версии **new**, если наряду с ней возможно использование перегруженных операций. Элемент **размещение** используется (если он предусмотрен перегруженной версией) для дополнительной информации о месте размещения в памяти. **Инициализатор** задает начальное значение создаваемого объекта.

Таким образом, обязательно должен быть указан только тип данных. Например:

```
double *A = new double;
```

В данном случае в памяти динамически создается объект — действительное число. В дальнейшем доступ к нему осуществляется как ***A**. Например:

```
*A = 5.1;
Label1->Caption = *A;
```

Если нет желания вводить указатель на объект и в дальнейшем работать с этим указателем, можно динамически разместить объект с помощью следующего оператора:

```
double B = *new double;
```

В этом случае в дальнейшем на объект можно ссылаться просто по имени — **B**.

Создание динамически размещенного объекта можно совместить с его инициализацией. Например:

```
double *A = new double(5.1);
double B = *new double(5.5);
```

Ниже приведен пример создания и динамического размещения в памяти компонента — окна редактирования типа **TEdit**:

```
TEdit *Edit = new TEdit(this);
Edit->Parent = Form1;
```

Первый оператор выделяет память под объект и создает его, передавая в него указатель **this** как владельца **Owner**. Второй задает для компонента родителя — **Form1**. В этот момент компонент станет виден на форме.

Рассмотрим подробнее выполнение операции **new** при создании и размещении в памяти объекта. Операция определяет объем необходимой памяти, используя неявно операцию **sizeof(тип)**. Если в динамически распределяемой области памяти есть место для размещения объекта, то выделяется соответствующий блок памяти и операция **new** возвращает указатель на объект данного типа. При этом нет необходимости явно приводить тип этого указателя — все делается автоматически. Созданный объект хранится в памяти, пока не будет уничтожен описанной далее операцией **delete** или пока не завершится выполнение программы.

Если в памяти невозможно выделить блок требуемого размера, генерируется исключение **bad_alloc**. Поэтому в программе всегда надо предусматривать блок **catch** (см. раздел 12.10.5), который бы перехватывал это исключение прежде, чем программа попытается получить доступ к создаваемому объекту. Таким образом, динамическое размещение объектов в памяти, как правило, должно оформляться следующим образом:

```
#include <iostream.h>
try
{
    Операторы динамического распределения памяти с помощью new
}
catch(std::bad_alloc)
{
    Операторы действий при недостаточной памяти
}
```

Можно отменить генерацию исключения **bad_alloc**, задавая указатель на свой собственный обработчик событий, связанных с невозможностью выделить память. Для этого используется оператор

```
set_new_handler(указатель)
```

который позволяет задать указатель на обработчик. При этом **set_new_handler** возвращает прежний указатель, который был зарегистрирован до этого.

Например, вы можете описать функцию

```
void F1(void)
{
    ShowMessage("Не хватает памяти");
    exit(1);
}
```

которая обрабатывает ситуацию, связанную с нехваткой памяти, и ввести в программу (например, в обработчик события **OnCreate** формы) оператор

```
set_new_handler(F1);
```

Вводимый таким образом обработчик не может ничего возвращать и должен или освободить память для выполнения **new**, или сгенерировать исключение **bad_alloc**, или завершить программу (это сделано в приведенном примере). Если не выполнено ни одно из этих действий, возникнет бесконечный цикл обращений к обработчику.

Можно отменить генерацию исключения **bad_alloc**, не вводя специального обработчика, а просто записав оператор

```
set_new_handler(0);
```

В этом случае при недостатке памяти операция **new** будет возвращать **NULL**. Тогда проверку можно строить, проверяя, не равен ли значению **NULL** указатель, возвращенный **new**.

Приведенные ранее примеры относились к динамическому размещению в памяти одиночных объектов. Аналогичным образом можно размещать и массивы. Например, оператор

```
double *A = new double[100];
```

динамически размещает массив из 100 действительных чисел. К его элементам в дальнейшем можно обращаться как обычно, по индексу: **A[ind]**. При использовании для создания массива операции **new** надо иметь в виду, что в момент создания его нельзя инициализировать, как это делается с одиночными объектами.

Можно создавать и многомерные массивы. Например, оператор

```
double *M = new double[100][100];
```

создает и динамически размещает в памяти двумерный массив. При размещении многомерных массивов надо иметь в виду, что первый размер можно задавать переменной, но остальные размеры задаются только константами. Например:

```
double *M = new double[n][100];
```

Динамически распределенную память надо освобождать, когда отпадает необходимость в размещенных в ней объектах. В противном случае получится неоправданная утечка памяти. Освобождение памяти осуществляется операцией **delete**. Она выполняет то же, что описанная ранее стандартная библиотечная функция **free**. Но использование **delete** предпочтительнее. Во всяком случае все, что размещается в памяти операцией **new**, должно удаляться операцией **delete**.

Операция может иметь следующие формы записи:

```
<::> delete <выражение>
<::> delete [ ] <выражение>
delete <имя_массива> [ ];
```

Например:

```
double *A = new double(5.1);
...
delete A;
```

или

```
double *A = new double[100];
...
delete [] A;
```

Операция **delete** освобождает память, но сама не задает указателю на эту память значения **NULL**. Поэтому желательно это делать программно, чтобы случайно в дальнейшем не воспользоваться указателем, который уже ни на что не указывает:

```
delete A;
A = NULL;
```

12.10 Исключения

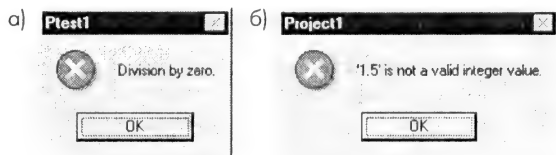
12.10.1 Исключения и их стандартная обработка

При работе программы могут возникать различного рода ошибки: переполнение, деление на ноль, попытка открыть несуществующий файл и т.п. При возникновении таких исключительных ситуаций программа генерирует так называемое *исключение* и выполнение дальнейших вычислений в данном блоке прекращается. Исключение — это объект специального вида, характеризующий возникшую в программе исключительную ситуацию. Он может также содержать в виде параметров некоторую уточняющую информацию. Особенностью исключений является то, что это сугубо временные объекты. Как только они обработаны каким-то обработчиком, они разрушаются.

Если исключение не перехвачено нигде в программе (как это делать — будет рассказано в последующих разделах), то оно обрабатывается методом **Application->HandleException**. Он обеспечивает стандартную реакцию программы на большинство исключений — выдачу пользователю краткой информации в окне сообщений и уничтожение экземпляра исключения. На рис. 12.1 приведены примеры таких стандартных сообщений для случаев целочисленного деления на ноль и попытки преобразовать функцией **StrToInt** строку «1.5» в целое число.

Рис. 12.1.

Примеры стандартных сообщений об ошибках деления на ноль (а) и преобразования (б)



Предупреждение

Если вы работаете в среде разработки C++Builder и отлаживаете свою программу, то при исключениях, помимо указанных на рис. 12.1 сообщений, могут появляться сообщения отладчика C++Builder, которые могут мешать вашей работе. Если хотите, то можете отключить появление этих сообщений. О том, как это делается, см. в главе 14 в разделе 14.2.8.

Если не принять соответствующих мер, то к неприятностям прекращения вычислений могут добавиться еще неприятности, связанные с так называемой *утечкой ресурсов*. Под этим подразумеваются потери динамически распределяемой па-

мяти, незакрытые файлы, не уничтоженные временные файлы на диске и прочий «мусор». Например, пусть вы выполняете некоторую программу, в которой имеются следующие операторы:

```
FILE *fp;
int size;
char *str;

...
fp = fopen("a.tmp", "w");
fprintf(fp, "файл a.tmp");
str = (char *) malloc(size);
```

<операторы, в которых может обнаружиться исключительная ситуация>

```
remove("a.tmp");
free(str);
```

Вы открываете временный файл (см. раздел 13.9.2) с именем **a.tmp**, чтобы хранить в нем какие-то промежуточные данные вычислений. В конце работы вы намерены уничтожить его процедурой **remove**. Вы динамически выделяете (см. раздел 12.9) некоторую память процедурой **malloc**, намереваясь освободить ее, когда она вам больше не будет нужна, процедурой **free**. Но если в промежуточных операторах возникнет исключение, то вычисления прервутся и процедуры **remove** и **free** не будут выполнены. В результате память, выделенная процедурой **malloc**, останется недоступной, а на диске сохранится временный и уже ненужный файл **a.tmp**.

Помимо указанного, стандартная обработка исключений программой имеет еще один недостаток — пользователь остается в полном недоумении, что же ему дальше делать? И не только не очень квалифицированный пользователь, которого приведенные на рис. 12.1 сообщения на английском языке могут повергнуть в шок. Даже опытному человеку невозможно порой догадаться, что же в вашей программе делится на нуль и как этого можно избежать. Наверное, каждый попадал в подобные ситуации, даже применяя профессионально сделанные программы, включая Windows.

Хороший стиль программирования

Программист должен принять все мыслимые меры, чтобы ни при каких ошибках пользователя и ни при каких сочетаниях данных приложение не заканчивалось бы аварийно. Но если все-таки аварийное завершение происходит, необходима полная зачистка «мусора» — удаление временных файлов, освобождение памяти, разрыв связей с базами данных и т.д.

12.10.2 Способы защиты кодов зачистки — блоки **try ... __finally** и функции **exit**

Рассмотрим способы защиты кодов зачистки «мусора». Первый из них — использование блока **try ... __finally**. Блок, содержащий совокупность операторов, способных привести к исключению, можно оформить следующим образом:

```
try
{
    // операторы, способные привести к исключению
}
__finally
{
    // операторы, выполняемые в любом случае
}
```

В этом случае операторы в разделе `__finally` будут выполняться всегда, независимо от того, было или не было исключение. Если было исключение, то после выполнения этих операторов вычисления, как и ранее, прерываются и возникает сообщение об исключении; в противном случае управление передается операторам, следующим за разделом `__finally`.

В качестве примера рассмотрим приведенный ранее фрагмент кода с временным файлом и динамическим распределением памяти, оформленный следующим образом:

```
FILE *fp;
int size;
char *str;
...
try
{
    fp = fopen("a.tmp", "w");
    fprintf(fp, "файл a.tmp");
    str = (char *) malloc(size);
    // операторы, в которых может обнаружиться исключительная ситуация
}
__finally
{
    remove("a.tmp");
    free(str);
}
```

В этом случае процедуры `remove` и `free` будут выполнены независимо от того, сгенерировано ли исключение в операторах блока `try`, или все вычисления в них закончились благополучно. Таким образом проблема зачистки «мусора» снимается — память в любом случае будет освобождена, а временный файл будет удален. Причем, это достигается ничтожным дополнительным кодом по сравнению с глобальной предварительной проверкой всех операций.

К сожалению, остаются другие из рассмотренных проблем: необходимость принять какие-то меры для дальнейшей нормальной работы программы при генерации исключения, а также необходимость уведомить пользователя о желательных действиях с его стороны (сообщения типа приведенных на рис. 12.1 в этом случае отображаются на экране, но они мало информативны для пользователя). Решить эти проблемы в данном случае невозможно, поскольку при выполнении операторов раздела `__finally` программа не знает, произошло ли исключение, и если произошло, то какое именно. Проверки наличия исключения с помощью функций `ExceptAddr` и `ExceptObject`, специально предназначенных для этого, внутри раздела `__finally` ни к чему не приводят, так как исключение генерируется после выполнения этих операторов.

Рассмотренные выше меры направлены на защиту кода зачистки в блоке. Однако, не все можно сделать на уровне блока. Поэтому полезно предусмотреть зачистку при завершении приложения.

Один из способов завершения приложения — вызов функции `exit`:

```
#include <stdlib.h>
void exit(int status);
```

Параметр `status` определяет код завершения. Обычно 0 соответствует нормальному завершению, а значение, отличное от нуля — аварийному при наличии ошибки выполнения. Можно, но не обязательно, использовать для задания значения `status` предопределенные константы: `EXIT_FAILURE` — аварийное завершение, `EXIT_SUCCESS` — нормальное.

Например:

```
exit(0); // нормальное завершение
exit(EXIT_SUCCESS); // нормальное завершение
```

```
exit(1);           // аварийное завершение
exit(EXIT_SUCCESS); // аварийное завершение
```

При завершении приложения с помощью **exit** перед прекращением работы закрываются все открытые файлы и очищаются все буфера вывода (печатается находящийся в них текст). Эти операции совершаются по умолчанию. Если же вам надо произвести еще какие-то действия (например, уничтожить временные файлы на диске), то вы можете зарегистрировать одну или несколько собственных функций, которые всегда автоматически будут выполняться перед действиями по умолчанию.

Регистрируются собственные функции завершения с помощью функции **atexit**:

```
#include <stdlib.h>
int atexit(void (_USERENTRY * func)(void));
```

Здесь **func** — имя регистрируемой функции. Можно выполнить несколько вызовов **atexit**, зарегистрировав таким образом несколько функций завершения. При завершении приложения выполняться эти функции будут в обратной последовательности: сначала — последняя из зарегистрированных, а в конце — зарегистрированная первой.

Например, следующий код определяет две функции завершения — **myexit1** и **myexit2**:

```
void myexit1(void)
{
    // операторы зачистки
}
void myexit2(void)
{
    // операторы зачистки
}
```

Эти функции могут не объявляться в заголовочном файле, а просто включаться в текст модуля.

Следующие операторы регистрируют эти функции:

```
atexit(myexit1);
atexit(myexit2);
```

Они могут быть включены, например, в обработчик события **OnCreate** главной формы приложения. Тогда при выполнении в любой точке программы вызова функции **exit** выполнятся операторы функции **myexit2**, затем операторы функции **myexit1**, затем выполнится зачистка по умолчанию (закрытие файлов и буферов), после чего произойдет завершение приложения.

Приведем пример функции **myexit1**, удаляющей в рабочем каталоге все временные файлы с расширением **.tmp** (использованные в примере функции **findfirst**, **findnext** и **remove** см. в главе 15 в разделе 15.5.6):

```
void myexit1(void)
{
    struct ffblk ffbk;
    int D;
    D = findfirst("*.tmp", &ffbk, 0);
    while (!D)
    {
        remove(ffbk.ff_name);
        D = findnext(&ffbk);
    }
}
```

Возможность ввести в процесс собственные функции зачистки делает завершение приложения вызовом **exit** «мягким» по сравнению с некоторыми другими способами.

Если приложение завершается закрытием главной формы методом **Close**, то коды зачистки можно вставить в обработчики событий, происходящих при выполнении этого метода. Таким образом это тоже «мягкий» способ завершения приложения. Причем, он имеет дополнительные преимущества, так как позволяет проанализировать ситуацию и в зависимости от каких-то условий завершить приложение, или не завершать его.

12.10.3 Иерархия классов исключений VCL

Для дальнейшего рассмотрения работы с исключениями надо представлять, хотя бы в первом приближении, иерархию классов объектов исключений и свойства этих объектов. Ниже приведена таблица иерархии большинства predefined в C++Builder классов исключений с краткими пояснениями. Создаваемые пользователем новые классы должны быть производными от одного из классов этой иерархии. Следует отметить, что помимо исключений, наследующих базовому классу **Exception** и используемых в объектах (компонентах) библиотеки VCL, имеется еще класс **exception**, стандартный для C++. Этот класс мы рассматривать не будем.

Exception	Базовый класс исключений VCL
EAbort	«Молчаливое» исключение, предназначенное для намеренного прерывания вычислений и быстрого выхода из глубоко вложенных процедур и функций
EAbstractError	Попытка вызвать абстрактный метод
EArrayError	Ошибка манипулирования с потомками класса TBaseArray : использование ошибочного индекса элемента массива, добавление слишком большого числа элементов в массив фиксированной длины, попытка вставки элемента в отсортированный массив
EAssertionFailed	Ложное выражение, проверяемое процедурой Assert в объектах VCL или в модулях Pascal
EBitsError	Ошибка доступа к массиву булевых величин TBits
ECacheError	Ошибка построения кэша в кубе решений
ECommonCalendarError	Ошибки ввода в компоненты, наследующие классу TCommonCalendar
EDateTimeError	Ошибка ввода даты или времени в компоненте TDateTimePicker
EComponentError	Ошибка регистрации или переименования компонентов
EConvertError	Ошибка преобразования строк или объектов (в частности, в функциях StrToInt , StrToFloat , StrToDate)
EDatabaseError	Ошибка работы с базами данных
EDBClient	Ошибка в наборе данных клиента. Свойство ErrorCode содержит код ошибки, возвращаемый BDE
EReconcileError	Ошибка обновления данных компонента TClientDataSet ; свойство Context содержит информацию в виде сообщения об ошибке, а свойство ErrorCode содержит код ошибки, возвращаемый BDE

EDBEngineError	Ошибка в BDE. Свойство Errors содержит информацию об ошибке — объект типа TDBErrors . Свойство ErrorCount хранит число ошибок
ENoResultSet	Генерируется компонентом TQuery при попытке открыть запрос без оператора SELECT
EUpdateError	Ошибка при обновлении в TProvider
EDBEditError	Ошибка при попытке приложения использовать данные, не соответствующие заданной маске для поля
EDimensionMapError	Ошибка формата данных в кубе решений
EDimIndexError	Ошибочный индекс в задании размерности в кубе решений
EExternal	Класс, перехватывающий исключения Windows
EAccessViolation	Ошибочный доступ к памяти; генерируется при попытке разыменования нулевого указателя NULL , попытке записи в кодовую страницу, попытке доступа к адресу вне памяти, распределенной приложению
EControlC	Нажатие пользователем клавиш Ctrl+C при выполнении консольного приложения. При обработке этого исключения можно выдать запрос пользователю, действительно ли он хочет прервать работу, и предпринять действия в зависимости от его ответа
EIntError	Базовый класс исключений целочисленных математических операций
EDivByZero	Попытка целочисленного деления на нуль
ERangeError	Целочисленное значение или индекс вне допустимого диапазона; используется только в Object Pascal
EIntOverflow	Переполнение при операции с целыми числами
EMathError	Базовый класс исключений операций с плавающей запятой; всегда генерируются только потомки этого исключения; обработка исключения EMathError может использоваться для перехвата всех исключений операций с плавающей запятой
EInvalidArgument	Недопустимое значение параметра при обращении к математической функции
EInvalidOp	Неопределенная операция с плавающей запятой: процессор наталкивается на неопределенную инструкцию, ошибочную операцию или переполняется стек процессора с плавающей запятой
EOverflow	Переполнение регистра при операциях с плавающей запятой
EUnderflow	Потеря значащих разрядов при выполнении операции с плавающей запятой
EZeroDivide	Деление на нуль числа с плавающей запятой

EPrivilege	Попытка приложения выполнить инструкцию процессора, которая недоступна для текущего уровня привилегий
EStackOverflow	Переполнение стека
EExternalException	Неизвестный код исключения
EHeapException	Ошибка динамического распределения памяти
EInvalidPointer	Ошибочная операция с указателем, например, попытка дважды освободить один и тот же блок памяти
EOutOfMemory	Неудачная попытка динамически выделить память; может генерироваться процедурой OutOfMemoryError
EOutOfResources	Генерируется при попытке приложения создать дескриптор Windows, когда Windows не имеет места для размещения дополнительных дескрипторов; возможно и при выделении других ресурсов Windows
EInOutError	Ошибка ввода-вывода из файла; исключение генерируется, если включена опция I/O checking на странице Pascal окна опций проекта; информация о конкретном виде ошибки (см. раздел 15.1.5.2) содержится в локальной переменной ErrorCode
EIntfCastError	Ошибочное применение операции преобразования типов интерфейса
EInvalidCast	Ошибка преобразования типа объекта
EInvalidGraphic	Нераспознаваемый графический файл
EInvalidGraphicOperation	Ошибочная операция с графикой, например, попытка изменить размер пиктограммы или копирование пиктограммы в буфер Clipboard
EInvalidGridOperation	Ошибочная операция с таблицей
EInvalidOperation	Ошибочная операция с компонентом; генерируется при попытке выполнить операцию, которая требует обработчика окна, над компонентом, не имеющем родителя (свойство Parent = NULL). Это исключение также генерируется при выполнении операций перетаскивания над формой (например, при попытке выполнить операцию Form1::BeginDrag).
EListError	Ошибка работы с объектом типа списка TStringList и TStrings : попытке сослаться на элемент с индексом вне допустимых пределов, попытке добавления дубликата строки в объект TStringList , в котором значение свойства Duplicates равно dupError , попытке вставить элемент в сортированный список, так как это может нарушить правильную последовательность элементов
ELowCapacityError	Попытка выделить памяти больше, чем доступно кубу решений; надо или увеличить значение Capacity , или уменьшить размерность куба
EMCIDeviceError	Ошибка доступа к устройствам мультимедиа через драйвер Media Control Interface (MCI)

EMenuError	Ошибка, связанная с элементами меню
EOleCtrlError	Генерируется при невозможности связать приложение с компонентом ActiveX
EOleError	Низкоуровневая ошибка OLE; C++Builder проверяет это исключение, но не генерирует его
EOleSysError	Ошибка OLE, специфическая для интерфейса OLE IDispatch; свойство ErrorCode содержит номер ошибки
EOleException	Ошибка OLE, связанная с методом или свойством
EOutlineError	Ошибка при работе с компонентом Outline
EOutOfResources	Генерируется при попытке приложения создать дескриптор Windows, когда Windows не имеет места для размещения дополнительных дескрипторов; возможно и при выделении других ресурсов Windows
EPackageError	Исключение времени проектирования, генерируемое при загрузке или использовании пакета
EParserError	Ошибка преобразования текста описания формы в двоичное представление, происходящая обычно из-за синтаксической ошибки исходного текста (часто из-за исправления текста вручную)
EPrinter	Ошибка печати; например, приложение пытается использовать принтер, которого нет, или задание по какой-то причине не может быть послано на принтер
EPropReadOnly	Попытка записать с помощью автоматизации OLE значение свойства, которое предназначено только для чтения
EPropWriteOnly	Попытка прочитать с помощью автоматизации OLE значение свойства, которое предназначено только для записи
EPropertyError	Ошибка при задании значения свойства
ERegistryException	Ошибка при обращении к реестру
EResNotFound	Ошибка при загрузке файла ресурсов .DFM или .RES в процессе проектирования.
EStreamError	Базовый класс исключений ошибок потоков
EFCreateError	Ошибка создания файла; например, пользователь указал недопустимое имя файла или указанный файл уже существует и не может быть перезаписан, так как пользователь не обладает соответствующим уровнем доступа
EFOpenError	Ошибка открытия файл
EFilerError	Базовый класс исключений файловых потоков
EReadError	Невозможно прочесть заданное число байтов
EWriteError	Невозможно записать заданное число байтов
EClassNotFound	Компонент не связан с приложением

EInvalid-Image	Невозможно прочесть файл ресурсов
EStringListError	Ошибочный доступ к окну списка с неверным индексом
EThread	Конфликт в многопоточном приложении (например, вызов метода Synchronize объекта TThread до успешного завершения его предыдущего вызова)
ETreeViewError	Ошибка индекса при работе с компонентом TTreeView
EUnsupportedTypeError	Ошибка выбора типа поля в качестве размерности куба решений
EVariantError	Ошибка, связанная с типом данных Variant
EWin32Error	Ошибка Windows, генерируется процедурой Raise-LastWin32Error , если Windows возвращает ошибку

12.10.4 Базовый класс исключений VCL Exception

Все предопределенные в C++Builder классы исключений, как видно из их иерархии, приведенной в разделе 12.10.3, являются прямыми или косвенными наследниками класса **Exception**, объявленного в модуле SysUtils и наследующего непосредственно **TObject**.

12.10.4.1 Свойства исключений

В классе **Exception** объявлено два свойства:

Свойство	Тип	Описание
HelpContext	int	Целый идентификатор экрана контекстно-зависимой справки. Этот экран справки отображается, если пользователь, находясь в окне с сообщением об ошибке, нажимает клавишу F1. По умолчанию значение равно 0
Message	System::AnsiString	Строка сообщения, которая в дальнейшем при обработке исключения системным обработчиком отображается в окне сообщений; устанавливается конструктором с умолчанием

Свойство **Message** имеет значение по умолчанию, которое присваивается при автоматической генерации исключения. При преднамеренной генерации исключений их конструкторы, описанные в следующем разделе, могут задавать значение свойства **Message** в виде переменной типа **string** или литеральной константы.

Свойство **HelpContext** хранит целый идентификатор экрана контекстно-зависимой справки. Этот экран справки отображается, если пользователь, находясь в окне с сообщением об ошибке, нажимает клавишу F1.

По умолчанию значение свойства **HelpContext** равно 0. Это значение может изменяться некоторыми конструкторами (см. следующий раздел). Например, оператор

```
throw Exception("Не хватает исходных данных", 4);
```

генерирует исключение со значением свойства **Message**, равным тексту «Не хватает исходных данных», и значением свойства **HelpContext**, равным 4. При получении

нии сообщения об этом исключении пользователь сможет нажать клавишу F1 и получить пояснения, что ему делать в этом случае.

Конечно, чтобы это работало, надо создать соответствующий файл справки и связать его с приложением, установив соответствующую опцию Help file (файл справки) в окне Project Options (опции проекта) на странице Application (приложение). Разработка файла справки подробно рассмотрена в главе 8, а связь приложения с файлом справки — в разделе 4.1.9.

12.10.4.2 Конструкторы исключений

Класс **Exception** наследует все функции своего базового класса **TObject**, в частности, полезную для идентификации неизвестного исключения функцию **ClassName**.

Кроме того, в интерфейсе класса **Exception** описаны 8 конструкторов, наследуемых всеми исключениями:

Конструктор	Описание
Exception(const System::AnsiString Msg)	Конструктор, передает строку сообщения Msg свойству Message
Exception(const System::AnsiString Msg, const System::TVarRec * Args, const int Args_Size)	Конструктор формирует строку свойства Message , исходя из строки описания формата Msg и массива аргументов Args размером Args_Size
Exception(int Ident)	Конструктор задает строку свойства Message идентификатором Ident строки сообщения в ресурсах проекта
Exception(int Ident, const System::TVarRec * Args, const int Args_Size)	Конструктор задает строку свойства Message идентификатором Ident строки описания формата в ресурсах проекта и массивом аргументов Args
Exception(const System::AnsiString Msg, int AHelpContext)	Конструктор передает строку сообщения Msg свойству Message ; передает свойству HelpContext идентификатор HelpContext экрана контекстно-зависимой справки по этому исключению
Exception(const System::AnsiString Msg, const System::TVarRec * Args, const int Args_Size, int AHelpContext)	Конструктор формирует строку свойства Message , исходя из строки описания формата Msg и массива аргументов Args ; передает свойству HelpContext идентификатор HelpContext экрана контекстно-зависимой справки по этому исключению
Exception(int Ident, int AHelpContext)	Конструктор задает строку свойства Message идентификатором Ident строки сообщения в ресурсах проекта; передает свойству HelpContext идентификатор HelpContext экрана контекстно-зависимой справки по этому исключению

```
Exception(int Ident,
          const System::TVarRec * Args,
          const int Args_Size,
          int AHelpContext)
```

Конструктор формирует строку свойства **Message** исходя из строки описания формата в ресурсах проекта, указываемой идентификатором **Ident**, и массива аргументов **Args**; передает свойству **HelpContext** идентификатор **HelpContext** экрана контекстно-зависимой справки по этому исключению

Рассмотрим примеры использования различных конструкторов:

```
throw Exception("Не хватает исходных данных");
```

```
throw Exception(Format("Задано %d параметров из %d",
                      OPENARRAY(TVarRec, (N1, N2))));
```

Последний пример использует функцию **Format** (см. раздел 15.3.1.2 главы 15) для форматированного вывода информации о значениях переменных **N1** и **N2**. При этом для передачи в конструктор массива используется макрос **OPENARRAY** (о передаче в функции открытых массивов см. в главе 13 в разделе 13.10.3). В результате, например, при значениях переменных **N1 = 5** и **N2 = 7** будет сгенерировано исключение, в диалоговом окне которого появится текст: «Задано 5 параметров из 7».

Следующий пример:

```
throw Exception("Задано %d параметров из %d",
                OPENARRAY(TVarRec, (N1, N2)));
```

Этот пример аналогичен предыдущему, но использует конструктор с непосредственным заданием строки форматирования в качестве первого параметра. Поэтому запись получается несколько короче, чем в предыдущем примере.

Следующий пример генерирует исключение с указанием темы контекстно зависимой справки:

```
throw Exception("Не хватает исходных данных", 4);
```

Этот оператор сгенерирует исключение с тем же текстом, что и в одном из приведенных выше примеров, но если в диалоговом окне с сообщением об этом исключении пользователь нажмет клавишу **F1**, ему будет предъявлена контекстная справка с идентификатором 4.

Пример применения конструктора, использующего строку ресурсов:

```
throw Exception(65369);
```

Этот оператор передает в свойство **Message** строку с номером 65369 из файла ресурсов. Оператор

```
raise EMy.CreateResFmt(65369, OPENARRAY(TVarRec, (N1, N2)));
```

берет из файла ресурсов строку с номером 65369 как строку описания формата и передает в свойство **Message** сформатированные с ее помощью значения переменных **N1** и **N2**.

12.10.5 Обработка исключений в блоках try ... catch

12.10.5.1 Синтаксис блоков try ... catch

Наиболее кардинальный путь борьбы с исключениями — отлавливание и обработка их с помощью блоков **try ... catch**. Синтаксис этих блоков следующий:

```

try
{
    Исполняемый код
}
catch ( TypeToCatch )
{
    Код, исполняемый в случае ошибки
}

```

Операторы блока **catch** представляют собой обработчик исключения. Параметр **TypeToCatch** может быть или одним из целых типов (**int**, **char** и т.п.), или ссылкой на класс исключения, или многоточием, что означает обработку любых исключений. Смысл параметров целого типа будет рассмотрен ниже в разделе 12.10.6.1. А пока остановимся на случае, когда параметр является ссылкой на класс исключений.

Операторы обработчика выполняются только в случае генерации в операторах блока **try** исключения типа, указанного в заголовке **catch**. После блока **try** может следовать несколько блоков **catch** для разных типов исключений. Таким образом, в обработчиках **catch** вы можете предпринять какие-то действия: известить пользователя о возникшей проблеме и подсказать ему пути ее решения, принять какие-то меры к исправлению ошибки (например, при переполнении заслать в результат очень большое число соответствующего знака) и т.д. Наиболее ценным является то, что вы можете определить тип сгенерированного исключения и дифференцированно реагировать на различные исключительные ситуации. Причем перехват исключения блоком **catch** приводит к тому, что это исключение далее не обрабатывается стандартным образом, т.е. пользователю не предъявляется окно с непонятными ему английскими текстами.

Приведем пример обработки исключений. Пусть в вашем приложении имеется два окна редактирования **Edit1** и **Edit2**, в которых пользователь вводит действительные числа типа **float**. Приложение должно разделить их одно на другое. При этом возможен ряд ошибок: пользователь может ввести в окно символы, не преобразуемые в целое число, может ввести слишком большое число, может ввести вместо делителя нуль, результат деления может быть слишком большим для типа **float**. Следующий код отлавливает все эти ошибки:

```

float A;
try
{
    A = StrToFloat(Edit1->Text) / StrToFloat(Edit2->Text);
}
catch(EConvertError&)
{
    Application->MessageBox("Вы ввели ошибочное число",
                           "Повторите ввод", MB_OK);
}
catch(EZeroDivide&)
{
    Application->MessageBox("Вы ввели нуль",
                           "Повторите ввод", MB_OK);
}
catch(EOverflow&)
{
    Application->MessageBox("Переполнение",
                           "Ошибка вычислений", MB_OK);
    if (StrToFloat(Edit1->Text) * StrToFloat(Edit2->Text) >= 0)
        A = 3.4E38;
    else A = -3.4E38;
}

```

Если пользователь ввел неверное число (например, по ошибке нажал не цифру, а какой-то буквенный символ), то при выполнении функции **StrToFloat** возникнет исключение класса **EConvertError**. Соответствующий обработчик исключения сообщит пользователю о сделанной ошибке и посоветует повторить ввод. Аналогичная реакция последует на ввод пользователем в качестве делителя нуля (класс исключения **EZeroDivide**). Если возникает переполнение, то соответствующий блок **catch** перехватывает исключение, сообщает о нем пользователю и исправляет ошибку: заносит в результат максимально возможное значение соответствующего знака.

Поскольку исключения образуют иерархию, рассмотренную в разделе 12.10.3, можно обрабатывать сразу некоторую совокупность исключений, производных от одного базового исключения. Для этого надо в заголовке блока **catch** указать имя этого базового исключения. Например, исключения **EZeroDivide** (целочисленное деление на нуль), **EOverflow** (переполнение при целочисленных операциях), **EInvalidArgument** (выход числа за допустимый диапазон) и некоторые другие являются производными от класса исключений **EMathError**. Поэтому все их можно отлавливать с помощью одного блока **catch**, например, такого:

```
catch (EMathError&)
{
    Application->MessageBox("Ошибка вычислений",
                           "Повторите ввод", MB_OK);
}
```

Правда, в этом случае не конкретизируется причина прерывания исключений. Однако, такая конкретизация возможна, если воспользоваться свойствами исключений. Все исключения имеют свойство **Message**, которое представляет собой строку, отображаемую пользователю при стандартной обработке исключений.

Чтобы воспользоваться свойствами исключений, надо в заголовке блока **catch** не только указать тип исключения, но и создать временный указатель на объект этого типа. Тогда через имя этого объекта вы получаете доступ к его свойствам. Ниже приведен пример использования свойств исключений при перехвате исключений, наследующих классу **EMathError**:

```
catch (EMathError& E)
{
    AnsiString S = "Ошибка вычислений : ";
    if(E.Message == "EZeroDivide") S += "деление на нуль";
    if(E.Message == "EOverflow") S += "переполнение";
    if(E.Message == "EInvalidArgument") S += "недопустимое число";
    Application->MessageBox(S.c_str(), "Повторите ввод", MB_OK);
}
```

Вводимое в этом операторе имя ссылки на исключение **E** носит сугубо локальный характер и вводится только для того, чтобы можно было сослаться на свойство **Message** по имени объекта исключения.

Как уже говорилось выше, если в заголовке блока **catch** указано многоточие, то этот блок перехватит любые исключения:

```
catch(...)
{
    ShowMessage("Призошла ошибка.");
}
```

Блок **catch(...)** может сочетаться и с другими блоками **catch**, но в этом случае он должен, конечно, располагаться последним. Поскольку этот блок перехватит все исключения, то все блоки, следующие за ним, окажутся недоступными. C++Builder следит за этим. Если блок **catch(...)** оказался не последним, вам будет выдано компилятором сообщение об ошибке с текстом: «The handler must be last» («Обработчик должен быть последним»).

Предупреждение

Следует указать на некоторую опасность применения блока `catch(...)`. Перехват всех исключений способен замаскировать какие-то непредвиденные ошибки в программе, что затруднит их поиск и снизит надежность работы.

12.10.5.2 Последовательность обработки исключений, обработка на уровне приложения

Блоки `try...catch` могут быть вложенными явным или неявным образом. Примером неявной вложенности является блок `try...catch`, в котором среди операторов раздела `try` имеются вызовы функций, которые имеют свои собственные блоки `try...catch`. Рассмотрим последовательность обработки исключений в этих случаях. При генерации исключения сначала ищется соответствующий ему обработчик в том блоке `try...catch`, в котором создалась исключительная ситуация. Если соответствующий обработчик не найден, поиск ведется в обрамляющем блоке `try...catch` (при наличии явным образом вложенных блоков) и т.д. Если в данной функции обработчик не найден или вообще в ней отсутствуют блоки `try...catch`, то поиск переходит на следующий уровень — в блок, из которого была вызвана данная функция. Этот поиск продолжается по всем уровням. И только если он закончился безрезультатно, выполняется стандартная обработка исключения, заключающаяся, как уже было сказано, в выдаче пользователю сообщения о типе исключения.

Как только блок `catch`, соответствующий данному исключению, найден и выполнен, объект исключения разрушается и управление передается оператору, следующему за соответствующим блоком `try...catch`.

Возможен также вариант, когда в самом обработчике исключения в процессе обработки возникла исключительная ситуация. В этом случае обработка прерывается, прежнее исключение разрушается и генерируется новое исключение. Его обработчик ищется в блоке `try...catch`, внешнем по отношению к тому, в котором возникло новое исключение.

Если исключение не перехвачено ни одним обработчиком в функциях, вы можете обработать его на уровне приложения. Для этого предусмотрены события `OnException` компонента `Application` — самого приложения. Обработчик этих событий можно ввести в ваше приложение следующим образом. Пусть вы решили назвать этот обработчик `MyException`. Тогда в заголовочный файл приложения надо добавить его объявление:

```
void __fastcall MyException(TObject *Sender, Exception *E);
```

В файл вашего модуля надо внести реализацию обработчика:

```
void __fastcall TForm1::MyException(TObject *Sender, Exception *E)
{
    // Операторы обработки
}
```

Осталось указать приложению на вашу функцию `MyException` как на обработчик события `OnException`. Вы можете это сделать, включив, например, в обработку события формы `OnCreate` оператор:

```
Application->OnException = MyException;
```

Ваш обработчик не перехваченных ранее исключений готов. Осталось только наполнить его операторами, сообщающими пользователю о возникших неполадках и обеспечивающими дальнейшую работу программы. К вашей функции `MyException` приложение будет обращаться, если было сгенерировано исключение и ни один блок `catch` его не перехватил. В функцию передается указатель `E` на объект класса `Exception`. Этот объект является сгенерированным исключением, а класс `Exception` — базовый класс всех исключений.

Простейшая обработка исключения могла бы производиться функцией **ShowException**, обеспечивающей отображение информации об исключении:

```
Application->ShowException(E);
```

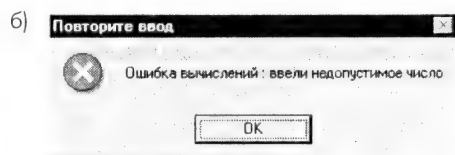
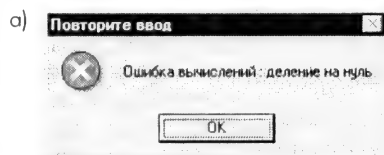
Примеры сообщений, выдаваемых этой функцией, были приведены ранее на рис. 12.1. В заголовке окна пишется имя приложения, а текст содержит описание причины генерации исключения. Основным недостатком функции являются сообщения на английском языке, что вряд ли порадует пользователей вашего приложения. Поэтому лучше сделать собственные сообщения. При этом для определения истинного класса сгенерированного исключения можно воспользоваться методом **ClassName**. Тогда обработчик события **OnException** может иметь, например, следующий вид:

```
void __fastcall TForm1::MyException(TObject *Sender, Exception *E)
{
    AnsiString S = "Ошибка вычислений : ";
    if (String(E->ClassName()) == "EZeroDivide")
        S += "деление на ноль";
    if (String(E->ClassName()) == "EOverflow")
        S += "переполнение";
    if (String(E->ClassName()) == "EInvalidArgument")
        S += "недопустимое число";
    if (String(E->ClassName()) == "EConvertError")
        S += "ввели недопустимое число";
    Application->MessageBox(S.c_str(), "Повторите ввод", MB_OK);
}
```

На рис. 12.2 приведены примеры сообщений, выдаваемых этим обработчиком. Вероятно, пользователям более понравятся сообщения рис. 12.2, чем сообщения рис. 12.1.

Рис. 12.2.

Сообщения, выдаваемые вашим обработчиком при делении на ноль (а) и при неверной записи вводимого числа (б)



12.10.6 Преднамеренная генерация исключений

12.10.6.1 Оператор throw

В ряде случаев возникает потребность сгенерировать исключение искусственно. Например, вы обработали какое-то исключение, но хотите, чтобы его обработка была завершена обработчиком внешнего по отношению к данному блоку **try...catch**. В этом случае вам надо повторно сгенерировать исключение того же типа, что и прежде, поскольку прежнее разрушено данным обработчиком.

Повторная генерация исключения осуществляется ключевым словом **throw**. Общая схема такой двухэтапной обработки исключений может иметь вид:


```

try
{
    // операторы внешнего блока
    try // начало внутреннего блока
    {
        // операторы внутреннего блока,
        // способные привести к генерации исключения
    }
    catch(тип исключения &)
    {
        // обработка исключения, сгенерированного во внутреннем блоке
        throw; // повторная генерация того же исключения
    }
    // операторы внешнего блока;
    // при генерации исключения не выполняются
} // завершение внешнего блока try
catch(тип исключения &)
{
    // обработка исключений и внутреннего, и внешнего блоков
}

```

При такой организации программы исключения, сгенерированные во внутреннем блоке, обрабатываются в два этапа: сначала обработчиком внутреннего блока, а затем обработчиком внешнего блока. При этом операторы внешнего блока, следующие за внутренним, при генерации исключения во внутреннем блоке выполняться не будут. Исключения, сгенерированные во внешнем блоке, будут обрабатываться только обработчиками этого внешнего блока.

С помощью ключевого слова **throw** можно сгенерировать не только повторное исключение, но и исключение любого типа в любом месте программы. Такая необходимость, в частности, возникает, когда пользователь что-то не так сделал и не имеет смысла продолжать выполнение приложения. Например, пользователь должен был задать какую-то информацию в окнах редактирования, но забыл это сделать. В этом случае прежде, чем продолжать работу, надо указать пользователю на его ошибку. Это можно сделать, сгенерировав соответствующее исключение.

Генерация нестандартного исключения производится ключевым словом **throw**, после которого указывается генерируемый объект любого типа. Это исключение может в дальнейшем перехватываться блоком **catch**, в заголовке которого указан то же тип, что у сгенерированного объекта. Например, вы можете написать оператор, который проверяет, задана ли информация в окне редактирования **Edit1**, и, если не задана, то генерируется исключение:

```
if(Edit1->Text == "") throw "Не задана требуемая информация";
```

В данном случае объект генерируемого исключения имеет тип **char ***. Подобных операторов с различными текстами в разных местах кода может быть много. И все они могут быть перехвачены, например, следующим блоком **catch**:

```

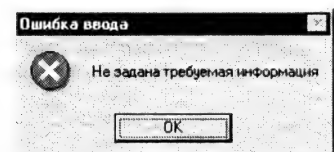
catch(char * s)
{
    Application->MessageBox(s, "Ошибка ввода", MB_ICONHAND|MB_OK);
}

```

Заголовок этого блока обеспечивает перехват любых исключений типа **char *** и отображение их текстов в диалоговом окне, пример которого показан на рис. 12.3.

Рис. 12.3.

Окно, отображающее текст перехваченного исключения типа **char ***



Вы можете в качестве параметра **throw** задавать целые числа, отображающие некие номера ошибок. Например:

```
if (...) throw 1;
```

В данном случае генерируется объект исключения типа **int**. Поэтому подобные исключения могут быть перехвачены и обработаны блоком вида:

```
catch(int& i)
{
    switch (i)
    {
        case 1: ...
            break;
        case 2: ...
            break;
        ...
    }
}
```

Можно генерировать объекты исключений и более сложных типов, например, структуры с полями, которые анализируются в обработчике исключения. Пусть, например, пользователь перед занесением в базу данных новой записи, относящейся к некоторому объекту, должен задать некоторый минимум параметров (характеристик) этого объекта. В приведенном ниже коде создается структура **st** типа **Pers** и в ходе диалога с пользователем в нее заносится число заданных параметров (**st.i1**) объекта. Тогда перед занесением в базу данных программа может сверить требуемое (**st.i1**) и действительно заданное (**st.i2**) число параметров. Если параметров задано недостаточно, то генерируется исключение, объектом которого является структура **st**. Обработчик этого исключения в свою очередь может проанализировать все поля структуры и выдать пользователю соответствующее сообщение.

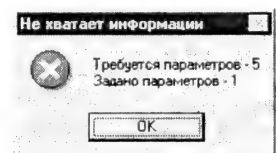
```
struct Pers
{
    int i1, i2;
} st = {0,5};

try
{
    if(...) st.i1++;
    ...
    if(st.i1 < st.i2) throw st;
}
catch (Pers&)
{
    Application->MessageBox(("Требуется параметров - " +
        IntToStr(st.i2) + "\nЗадано параметров - " +
        IntToStr(st.i1)).c_str(),
        "Не хватает информации", MB_ICONHAND | MB_OK);
}
```

Пример выдачи информации таким обработчиком исключения приведен на рис. 12.4.

Рис. 12.4.

Пример выдачи сообщения об объекте исключения в виде структуры



12.10.6.2 Исключение EAbort и функция Abort

В C++Builder имеется исключение **EAbort**, несколько отличающееся от рассмотренных ранее. Генерация этого исключения, как и любых других, прерывает процесс вычисления. Но если приложение не отлавливает соответствующим блоком **catch** исключений этого класса, то они попадают в обработчик **TApplication::HandleException** и там, в отличие от других исключений, разрушаются без всяких сообщений. Таким образом, это «молчаливое» прерывание процесса вычисления, при котором не должно отображаться диалоговое окно с сообщением об ошибке.

Простейший путь генерации исключения **EAbort** — вызов функции **Abort**. Например:

```
if (...) Abort();
```

Только нельзя путать две похожие внешне функции: **Abort** — генерация «молчаливого» исключения, и **abort** — аварийное завершение программы.

Обычное применение **EAbort** — прерывание вычислений при выполнении некоторого условия окончания или условия прерывания пользователем (например, при нажатии клавиши Esc или какого-то оговоренного сочетания клавиш). Функция **Abort** прерывает текущую процедуру и все вызвавшие ее процедуры, передавая управление на самый верх. Таким образом, это наиболее простой выход из глубоко вложенных процедур. Впрочем, можно при необходимости перехватить исключение на каком-то промежуточном уровне, предусмотрев на нем блок **try...catch** и вставив соответствующий оператор обработки:

```
catch(EAbort&)
{
    ...
}
```

12.11 Сигналы

Сигнал — это некоторое непредвиденное событие (прерывание), которое может вызвать преждевременное завершение программы. Перечислим некоторые из таких непредвиденных событий: прерывание программы, вызванное нажатием пользователем клавиш Ctrl+C, появление недопустимой команды, ошибочный доступ к памяти (нарушение сегментации), запрос от операционной системы о завершении работы, ошибка операций с вещественными числами (деление на ноль или перемножение слишком больших действительных чисел).

Ниже перечислены некоторые стандартные сигналы, определенные в заголовочном файле **signal.h** (полный список приведен в главе 15 в разделе 15.6.1).

SIGABRT	Аварийное завершение программы (например, в результате вызова функции abort). Действие по умолчанию — вызов _exit(3)
SIGFPE	Ошибка арифметической операции, например, деление на ноль или операция, вызвавшая переполнение. Действие по умолчанию — вызов _exit(1)
SIGINT	Получение интерактивного сигнала (например, прерывание Ctrl+C). Действие по умолчанию — прерывание INT 23h
SIGUSR1 , SIGUSR2 , SIGUSR3	Определенные пользователем (только в Win32) сигналы пользователя, генерируемые функцией raise . Действие по умолчанию — игнорирование сигнала

Библиотека обработки сигналов содержит функцию **signal**, перехватывающую сигналы. В функцию **signal** передаются два параметра: целочисленный номер сигнала и указатель на функцию обработки сигнала.

Обычно сигналы автоматически генерируются при возникновении соответствующих событий. Но программа может целенаправленно генерировать сигналы функцией **raise**, в которую передаются целочисленное значение номера сигнала.

Например, вы можете предусмотреть в своей программе обработчик некоторого вводимого вами сигнала **SIGUSR1**. Пусть вы дали ему имя **Handl_SIGUSR1**. Тогда где-то в программе (например, при обработке события **OnCreate** формы) вам надо установить в системе этот обработчик с помощью оператора

```
signal(SIGUSR1, Handl_SIGUSR1);
```

При этом не забудьте вставить в файл директиву

```
#include <signal.h>
```

Сам обработчик сигнала может иметь вид:

```
void Handl_SIGUSR1(int N)
{
    ...
    if(MessageDlg("Продолжать?", mtConfirmation,
                  TMsgDlgButtons() << mbYes << mbNo, 0) == mrYes)
        // повторная установка обработчика:
        signal(SIGUSR1, Handl_SIGUSR1);
    else exit(EXIT_SUCCESS);
}
```

Этот обработчик принимает одно целое значение, соответствующее номеру сигнала. В обработчике предусматриваются некоторые действия, необходимые при появлении данного сигнала. Затем, если выполнение программы должно продолжаться, надо повторно установить обработчик сигнала с помощью функции **signal**, как показано в приведенном примере. Если этого не сделать, то последующие события **SIGUSR1** не будут вызывать этот обработчик. После выполнения команды повторной установки обработчика сигнала управление автоматически передается в точку программы, в которой сигнал был обнаружен. В этом, в частности, коренное отличие сигналов от исключений.

Генерация сигнала **SIGUSR1** в необходимых местах программы осуществляется оператором

```
raise(SIGUSR1);
```

Рассмотренный вариант функции сейчас считается несколько устаревшим. Более современный вариант (подробнее о нем см. в главе 15 в разделе 15.6.1) выглядит следующим образом. Определите в программе тип указателя на функцию **fptr**:

```
typedef void (*fptr)(int);
```

Этот указатель используется в вызове функции **signal**:

```
signal(SIGFPE, (fptr)Handl_SIGFPE);
```

Весь остальной приведенный выше текст примера может не изменяться.

Глава 13

Типы данных в языке C++

13.1 Классификация типов данных, объявление типов

Все типы, используемые в C++Builder, можно разбить на четыре группы:

Aggregate		структуры данных
	Array	массивы
	struct	структуры
	union	объединения
	class	классы
Function		функции
Scalar		скалярные
	Arithmetic	арифметические
	Enumeration	перечислимые
	Pointer	указатели
	Reference	ссылки
void		отсутствие значения

Другой способ классификации типов связан с их разбиением на *основные* и *производные* типы. К *основным* относятся: **void**, **char**, **int**, **float** и **double**, а также их варианты с модификаторами **short** (короткий), **long** (длинный), **signed** (со знаком) и **unsigned** (без знака). Например, **unsigned char**, **unsigned int**, **signed int** (модификатор **signed** подразумевается по умолчанию и поэтому обычно не указывается).

Основные типы в C++ следующие:

Тип	Размер в байтах	Диапазон значений
char	1	от -128 до 126
unsigned char	1	от 0 до 255
short	2	от -32 768 до 32 767
unsigned short	2	от 0 до 65 535
enum	2	от -2 147 483 648 до 2 147 483 647
long	4	от -2 147 483 648 до 2 147 483 647
unsigned long	4	от 0 до 4 294 967 295

Тип	Размер в байтах	Диапазон значений
int	4	как в long
unsigned int	4	как в unsigned long
float	4	от $3.4 \cdot 10^{-38}$ до $3.4 \cdot 10^{38}$
double	8	от $1.7 \cdot 10^{-308}$ до $1.7 \cdot 10^{308}$
long double	10	от $3.4 \cdot 10^{-4932}$ до $1.1 \cdot 10^{4932}$
bool	1	true или false

Имеются также основные типы `__int8`, `__int16`, `__int32`, `__int64`, о которых подробнее см. в разделе 13.3.

Следует отметить, что в C++Builder, в отличие от некоторых других версий C++, булев тип **bool** реализован как отдельный тип, а не как псевдоним целого. Однако, это не мешает при желании использовать в логических выражениях целые значения вместо булевых. При этом значение 0 расценивается как **false**, а любое ненулевое значение — как **true**.

Производные типы включают в себя указатели и ссылки на какие-то типы, массивы каких-то типов, типы функций, классы, структуры, объединения. Эти типы считаются производными, поскольку, например, классы, структуры, объединения могут включать в себя объекты различных типов.

Можно выделить еще одну категорию типов — *порядковые*, в которых значения упорядочены и для каждого из них можно указать предшествующее и последующее. К ним относятся целые, символы, перечислимые типы.

Типы данных указываются при объявлении любых переменных и функций (см. разделы 12.4.1 и 12.5.1). Например:

```
double a = 5.4, b = 2;
int c;
void F1(double A);
```

Пользователь может вводить в программу свои собственные типы. Объявления типов могут делаться в различных местах кода. Место объявления влияет на область видимости или область действия так же, как и в случае объявления переменных (см. раздел 12.6).

Синтаксис объявления типа:

```
typedef определение_типа идентификатор;
```

Здесь идентификатор — это вводимое пользователем имя нового типа, а определение_типа — описание этого типа. Например, оператор

```
typedef double Ar[10];
```

объявляет тип пользователя с именем **Ar** как массив из 10 действительных чисел. В дальнейшем на этот тип можно сослаться при объявлении переменных. Например:

```
Ar A = {1,2,3,4,5,6,7,8,9,10};
```

Объявление типа с помощью **typedef** можно использовать и для создания нового типа, имя которого будет являться псевдонимом стандартного типа C++. Именно так в C++Builder многие встроенные типы компонентов Object Pascal приведены к типам, характерным для C++. Эти переопределения типов содержатся в файле **sysdefs.h**. Например:

```
typedef bool Boolean;
typedef int Integer;
typedef short Smallint;
typedef unsigned char Byte;
```

Ниже дается таблица соответствия типов Delphi (т.е. типов Object Pascal) и типов C++.

Delphi	Размер или значение	Соответствие C++	Реализация
ShortInt	целое 8 бит	signed char	typedef
SmallInt	целое 16 бит	short	typedef
LongInt	целое 32 бита	int	typedef
Byte	целое без знака 8 бит	unsigned char	typedef
Word	целое без знака 16 бит	unsigned short	typedef
Integer	целое 32 бита	int	typedef
Cardinal	целое без знака 32 бита	unsigned int	typedef
Boolean	true/false	bool	typedef
ByteBool	true/false или целое без знака 8 бит	unsigned char	typedef
WordBool	true/false или целое без знака 16 бит	unsigned short	typedef
LongBool	true/false или целое без знака 32 бита	BOOL (WinAPI)	typedef
AnsiChar	символ без знака 8 бит	char	typedef
WideChar	символ Unicode размером в слово	wchar_t	typedef
Char	символ без знака 8 бит	char	typedef
AnsiString	AnsiString Delphi	AnsiString	класс
String[n]	прежний стиль строк Delphi, n = 1...255 бит	SmallString<n>	шаблон класса
ShortString	прежний стиль строк Delphi, 255 бит	SmallString<255>	typedef
String	AnsiString Delphi	AnsiString	typedef
Single	число с плавающей запятой 32 бита	float	typedef
Double	число с плавающей запятой 64 бита	double	typedef
Extended	число с плавающей запятой 80 бит	long double	typedef
Real	число с плавающей запятой 32 бита	double	typedef
Pointer	родовой указатель 32 бита	void *	typedef
PChar	указатель на символы 32 бита	unsigned char *	typedef
PAnsiChar	указатель на символы ANSI 32 бита	unsigned char *	typedef
Comp	число с плавающей запятой 64 бита	Comp	класс
OleVariant	значение variant OLE	OleVariant	класс

13.2 Приведение типов

В арифметических выражениях, содержащих элементы различных арифметических типов, C++Builder в процессе вычислений автоматически осуществляет преобразование типов. Это стандартное преобразование всегда осуществляется по принципу: если операция имеет операнды разных типов, то тип операнда «младшего» типа приводится к типу операнда «старшего» типа. Иначе говоря, менее точный тип приводится к более точному. Например, если в операции участвует короткое целое и длинное целое, то короткое приводится к длинному; если участвует целый и действительный операнды, то целый приводится к действительному и т.д. Таким образом, после подобного приведения типов оба операнда оказываются одного типа. И результат применения операции имеет тот же тип.

Все это относится к арифметическим операциям, но не относится к операции присваивания. Присваивание сводится к приведению типа результата выражения к типу левого операнда. Если тип левого операнда «младше», чем тип результата выражения, возможна потеря точности или вообще неправильный результат.

Рассмотрим примеры неявного автоматического преобразования типов. В результате действия следующих операторов

```
double a = 5.4, b = 2;  
int c = a * b;
```

переменная **c** получит значение 10, хотя истинное значение должно быть равно 10.8. Это значение действительно будет вычислено в результате умножения **a * b**, но затем дробная часть будет отброшена, поскольку **c** — целая переменная.

Результатом выполнения операторов

```
int m = 1, n = 2;  
double A = m / n;
```

будет значение **A = 0**. Поскольку **m** и **n** — целые переменные, то деление **m / n** сведется к целочисленному делению с отбрасыванием дробной части, результат которого равен нулю.

Результат выполнения похожих на предыдущие операторов

```
int m = 1;  
double n = 2;  
double A = m / n;
```

даст правильный результат — **A = 0.5**. Поскольку в данном случае один из операндов операции деления имеет тип **double**, то тип другого, целого операнда будет тоже приведен к **double** и результат деления будет иметь тип **double**.

Еще один пример, который дает совершенно неверный результат:

```
double a = 300, b = 200;  
short c = a * b;
```

Если вы попытаетесь реализовать этот пример, то увидите, что переменная **c** получит значение -5536, вместо ожидаемого 60 000. Дело в том, что переменная типа **short** может хранить значение не больше, чем 32 767. Поскольку выражение в правой части приведенного оператора дает результат 60 000, то его присваивание переменной типа **short** дает совершенно неверное значение.

Как было видно из некоторых приведенных примеров, неявное автоматическое приведение типов не всегда дает желаемый результат. Это можно исправить, применив операцию явного приведения типов. Она записывается в виде

(тип)

перед той величиной, которую вы хотите привести к указанному типу. Вернемся к уже рассмотренному примеру

```
int m = 1, n = 2;
double A = m / n;
```

который давал неверное значение переменной **A**. Этот результат можно исправить, применив во втором операторе явное приведение типа:

```
double A = (double)m / n;
```

В этом случае переменная **m**, к которой применяется операция приведения типа, рассматривается как действительная величина типа **double**. Тогда и переменная **n** неявно приводится к типу **double**, так что деление осуществляется уже не с целыми, а с действительными числами. Результат получается правильным — 0.5.

Есть еще одна ситуация, которая требует явного приведения типов: в некоторых случаях компилятор не может выбрать среди перегруженных функций (о перегрузке функций см. раздел 12.5.7 главы 12), если под данный тип параметра подходит несколько из них. Если в C++Builder 4 вы напишете код

```
TPoint P;
P.x = 5;
P.y = 1;
Labell->Caption = "Координата x = " + IntToStr(P.x);
```

то получите сообщение компилятора об ошибке: «Ambiguity between '_fastcall Sysutils::IntToStr(__int64)' and '_fastcall Sysutils::IntToStr(int)'» (Неоднозначность применения функции IntToStr к параметрам типов __int64 и int). Компилятор, как Буриданов осел, остановился между двумя (в данном случае идентичными) возможностями и отказывается производить выбор. Помочь компилятору легко, применив в последнем из приведенных операторов явное указание типа **int**:

```
Labell->Caption = "Координата x = " + IntToStr((int)P.x);
```

Подобный текст компилятор обработает без проблем.

В C++Builder 5 компилятор более «интеллектуальный» и в приведенном примере в подобной помощи не нуждается. Но в некоторых других сложных случаях подобное явное приведение типов может потребоваться.

13.3 Арифметические типы данных

Арифметические типы данных — это целые и действительные типы.

К целым типам относятся **char**, **short**, **int** и **long** вместе с их вариантами **signed** — со знаком и **unsigned** — без знака. Из этих ключевых слов может формироваться множество целых типов данных. Многие из них являются синонимами друг друга, как следует из следующей таблицы.

Синонимы	Примечания
char, signed char	Синонимы, если умолчанием для char задано signed
unsigned char	
char, unsigned char	Синонимы, если умолчанием для char задано unsigned
signed char	
int, signed int	
unsigned, unsigned int	
short, short int, signed short int	

Синонимы	Примечания
unsigned short, unsigned short int	
long, long int, signed long int	
unsigned long, unsigned long int	

Спецификаторы **signed** и **unsigned** могут применяться только к **char**, **short**, **int**, **long**. Если тип обозначен просто как **signed** или **unsigned**, то подразумеваются соответственно **signed int** и **unsigned int**.

При отсутствии в указании типа спецификатора **unsigned** для целых типов подразумевается **signed**. Исключением из этого правила является тип **char**. C++Builder позволяет вам установить в качестве умолчания для **char** **signed** или **unsigned**. В этом случае, если вы пишете объявление

```
char ch;
```

оно воспринимается как

```
signed char ch;
```

Если же вы хотите объявить переменную типа **char** без знака, вы должны это сделать явно:

```
unsigned char ch;
```

Спецификаторы **long** и **short** могут использоваться только с **int**. Если тип обозначен просто как **long** или **short**, то подразумеваются соответственно **long int** и **short int**.

Объем памяти, занимаемый различными целыми типами, не лимитирован стандартом ANSI C. Указано только, что **short**, **int** и **long** должны образовывать неубывающую последовательность, т.е. **short** ≤ **int** ≤ **long**. Поэтому не исключается, что все три типа требуют одинакового объема памяти. Таким образом, объем памяти может меняться от одной платформы к другой и это надо учитывать, если хотите создавать переносимые программы.

Объемы памяти, занимаемые целыми типами в данной версии C++Builder для 32 разрядных программ, приведены в таблице в разделе 13.1. В частности, из этой таблицы вы можете увидеть, что **int** и **long** эквивалентны и занимают по 32 бита.

Типы со знаком используют старший бит для хранения знака: 0 — положительный, 1 — отрицательный.

Помимо рассмотренных выше имеются еще целые типы, имена которых начинаются с символов «**__int**», за которыми следует число бит. При записи констант этих типов можно использовать суффиксы **i** и **ui**, как показано в приведенной ниже таблице. Впрочем, эти же суффиксы можно использовать и при задании значений переменных других целых типов.

Типы	Суффикс	Пример	Память (биты)
__int8	i8	__int8 c = 127i8;	8
__int16	i16	__int16 s = 32767i16;	16
__int32	i32	__int32 i = 123456789i32;	32
__int64	i64	__int64 big = 12345654321i64;	64
unsigned __int64	ui64	unsigned __64 hugeInt = 1234567887654321ui64;	64

Основными типами данных для представления действительных чисел с плавающей запятой являются типы **float** и **double**. Первый из них размещается в 32 битах, второй — в 64. К типу **double** может применяться спецификатор **long**, который увеличивает размер памяти до 80 бит.

Диапазоны возможных значений и затраты памяти для действительных типов в данной версии C++Builder для 32 разрядных программ приведены в таблице в разделе 13.1. Стандарт ANSI C не накладывает никаких ограничений на способ реализации действительных чисел. Поэтому, если вы хотите делать переносимые программы, то не ориентируйтесь на тот или иной размер памяти для действительных типов, а используйте для определения этого размера операцию **sizeof** (см. раздел 12.7.9 главы 12).

13.4 Типы строк

13.4.1 Массивы символов

В C++ отсутствует специальный тип строки. Строки рассматриваются как массивы символов, оканчивающиеся нулевым символом ('**\0**'). Строка доступна через указатель на первый символ в строке. Значением строки является адрес ее первого символа. Таким образом, можно сказать, что в C++ строка является указателем — указателем на первый символ строки. В этом смысле строки подобны массивам, потому что массив тоже является указателем на свой первый элемент. Подробно о работе с массивами символов см. в разделе 13.10.1, посвященном массивам.

Строка может быть объявлена либо как массив символов, либо как переменная типа **char***. Каждое из двух приведенных ниже эквивалентных объявлений

```
char S[] = "строка";  
char *Sp = "строка";
```

присваивает строковой переменной начальное значение «строка». Первое объявление создает массив из 7 элементов **S** содержащий символы '**c**', '**t**', '**r**', '**o**', '**k**', '**a**' и '**\0**'. Второе объявление создает переменную указатель **Sp**, который указывает на строку с текстом «строка», лежащую где-то в памяти. Но в любом случае число хранимых символов на 1 больше числа значащих символов за счет окончательного нулевого символа.

Доступ к отдельным символам строки осуществляется по индексам, начинающимся с нуля. Например, **S[0]** и **Sp[0]** — первые символы объявленных выше строк, **S[1]** и **Sp[1]** — вторые и т.д.

В приведенных объявлениях длина строк определялась автоматически компилятором. Можно объявлять строковые переменные заданной длины. Например, оператор

```
char buff[100];
```

объявляет переменную **buff**, которая может содержать строку до 99 значащих символов плюс заключительный нулевой символ.

Для обработки строк имеется ряд библиотечных функций. Основные из них **strcat** — конкатенация (склеивание) двух строк, **strcmp** — сравнение двух строк, **strcpy** — копирование одной строки в другую, **strstr** — поиск в строке заданной подстроки, **strlen** — определение длины строки, **strupr** — преобразование символов строки к верхнему регистру, **sprintf** — построение строки по заданной строке форматирования и списку аргументов и ряд других функций. Все они подробно рассмотрены в главе 15 в разделе 15.4.2. А пока рассмотрим несколько примеров их применения.

Начнем с самого простого. Выше было приведено объявление массива символов **buff**. Как занести в него какой-то текст? Это можно сделать с помощью функции **strcpy**:

```
strcpy(buff, "Текст, копируемый в buff");
```

Эта функция копирует строку, являющуюся ее вторым параметром, в строку, являющуюся первым параметром, и возвращает указатель на результат копирования.

Теперь решим задачу посложнее. Пусть, например, мы хотим прибавить в конец текста строки **S1** текст, хранящийся в строке **S2**. Это можно сделать с помощью функции **strcat**:

```
char S1[20] = "текст 1", S2[10] = "текст 2";
strcat(S1, S2);
```

Обратите внимание на то, что размер первой строки выбран с запасом, чтобы в ней уместились оба текста. Если не задать в объявлении размер строки, то он определится по присваиваемому ей тексту и в ней не останется места для каких-то добавлений.

Функция **strcat** прибавляет к тексту строки, указанной ее первым параметром, текст строки, указанной вторым параметром, и возвращает указатель на первую строку. Последнее обстоятельство позволяет делать вложенные вызовы **strcat**, если надо склеить несколько текстов. Давайте несколько усложним задачу. Пусть мы хотим оставить в неприкосновенности обе строки, а в третьей строке **S** хотим получить склеенные тексты строк **S1** и **S2**, разделенные символом пробела. Это можно сделать следующими операторами:

```
char *S1 = "текст 1", *S2 = "текст 2", S[20];
strcat(strcat(strcat(S, S1), " "), S2);
```

Самый внутренний вызов **strcat** склеивает пустую строку **S** и строку **S1**. Он возвращает указатель на **S** и, значит, следующий вызов **strcat** склеивает текст, появившийся в **S**, со строковой константой, содержащей символ пробела. Функция **strcat** опять возвращает указатель на **S** и последний внешний вызов **strcat** добавляет к уже сформированной строке текст строки **S2**.

Приведенный код будет работать, если есть уверенность, что сначала текст в **S** отсутствует. Чтобы не зависеть от исходного текста в **S**, лучше вместо внутреннего вызова **strcat** применить функцию

```
strcpy(S, S1)
```

При анализе текстовых строк часто надо найти в одной из строк фрагмент текста, заданный в другой строке. Этот фрагмент, например, может быть некоторым ключевым словом, символом и т.п. Эту задачу позволяет решить функция

```
strstr(S1, S2)
```

которая ищет в строке **S1** первое вхождение текста строки **S2** и, если поиск прошел удачно, возвращает указатель на первый символ этого вхождения. Если же текст не был найден, возвращается ноль.

Теперь давайте решим более сложную задачу. Пусть нам надо найти в строке **S1** первое вхождение текста строки **S2** и, если поиск прошел удачно, то заменить найденный фрагмент на текст, содержащийся в строке **S3**. Иначе говоря, требуется произвести контекстную замену в **S1** текста **S2** на текст **S3**. Один из возможных вариантов решения этой задачи приведен ниже.

```
char S1[20], S2[20], S3[20], S[60], *St;
// операторы занесения текста в S1, S2, S3
...
St = strstr(S1, S2);
if (St)
{
```

```

*St = 0;
St += strlen(S2);
Labell->Caption = strcat(strcat(strcpy(S, S1), S3), St);
}
else Labell->Caption = "Текст не найден";

```

Помимо строк **S1**, **S2**, **S3** в этом коде объявлена строка **S**, являющаяся буфером, в который будет помещаться текст с произведенной в нем заменой. Объявлен также указатель на строку **St**. Он нам потребуется в качестве вспомогательной переменной.

Первый выполняемый оператор кода ищет с помощью функции **strstr** вхождение строки **S2** в строку **S1** и присваивает результат поиска переменной **St**. Если функция **strstr** вернула нуль (это эквивалентно **false**), то печатается сообщение «Текст не найден». Если же поиск прошел успешно, то осуществляются следующие операции. Сначала в символ, на который указывает **St**, засылается 0 — это эквивалентно нулевому символу. Таким образом выделяется первая часть строки **S1**, расположенная до заменяемого текста. Затем указатель **St** сдвигается на длину заменяемого текста, которая определяется функцией **strlen**. После этой операции **St** начинает указывать на первый символ в строке **S1** после заменяемого текста. Следующий оператор формирует в буфере **S** текст с заменой и отображает его в метке **Labell**. Формирование текста осуществляется вложенными вызовами функций **strcat** и **strcpy**. Сначала срабатывает вложенный вызов **strcpy**. Он копирует в **S** строку, на которую указывает **S1**. Но поскольку вместо первого символа заменяемого текста мы занесли нулевой символ, то скопирована будет только начальная часть строки **S1** до этого символа. Затем срабатывает вложенный вызов **strcat** и к тексту, сформированному в **S** добавляется строка **S3**. Последний внешний вызов **strcat** добавляет к сформированному тексту часть строки **S1**, расположенную после замененного фрагмента. Именно на эту часть строки указывает **St**.

Чтобы это стало понятнее, разберем пример. Пусть строка **S1** содержит текст «Маша ела кашу», строка **S2** содержит текст «ела», а строка **S3** — «съела». Значит строка **S1** представляет собой массив:

```
'М', 'а', 'ш', 'а', ' ', 'е', 'л', 'а', ' ', 'к', 'а', 'ш', 'у', '\0'
```

После выполнения функции **strstr** указатель **St** будет указывать на шестой символ — букву 'е'. После того, как в этот символ заносится нуль, строка **S1** имеет вид:

```
'М', 'а', 'ш', 'а', ' ', '\0', 'л', 'а', ' ', 'к', 'а', 'ш', 'у', '\0'
```

После изменения **Pt** он начинает указывать на девятый символ — пробел после слова «ела». После вызова **strcpy** в строку **S** копируется первая часть строки **S1**, завершающаяся нулевым символом:

```
'М', 'а', 'ш', 'а', ' ', '\0'
```

После вложенного вызова **strcat** к строке **S** добавляется текст строки **S3**:

```
'М', 'а', 'ш', 'а', ' ', 'с', 'ъ', 'е', 'л', 'а', '\0'
```

И после внешнего вызова **strcat** к **S** добавляется строка, на которую указывает **St**, т.е. часть строки **S1**, начинающаяся с пробела после «ела»:

```
М', 'а', 'ш', 'а', ' ', 'с', 'ъ', 'е', 'л', 'а', ' ', 'к', 'а', 'ш', 'у', '\0'
```

В качестве последнего примера рассмотрим использование функции **sprintf**. Пусть в приложении имеется окно редактирования **Edit1**, в котором пользователь вводит фамилию сотрудника, и компонент **CSpinEdit1** типа **TCSpinEdit**, в котором вводится год рождения. Вы хотите сформировать строку вида «Сотрудник ..., ... г.р.», в которой вместо точек должны подставляться введенные данные: фамилия и год. Это можно сделать следующим кодом:

```
#include <stdio.h>
char S[40];
sprintf(S, "Сотрудник %s, %i г.п.", Edit1->Text, CSpinEdit1->Value);
```

Первый аргумент функции **sprintf** — формируемая строка. Второй — строка форматирования (ее полное описание см. в главе 15 в разделе 15.1.4.1). Она указывает текст формируемой строки и содержит спецификаторы, записываемые после символа «%», которые указывают формат включения в строку аргументов, список которых расположен в вызове **sprintf** после строки форматирования. В данном случае первый из этих параметров — текст в окне **Edit1**, вводимый со спецификатором **%s**, что означает строку, а второй параметр — значение года в компоненте **CSpinEdit1**, вводимое со спецификатором **%i**, что означает целое число.

Мы рассмотрели применение основных библиотечных функций работы со строками. Более полное изложение этих функций вы найдете в главе 15 в разделе 15.4.2.

C++Builder не ограничивается изложенным выше типичным для C++ подходом, сводящим строки к массивам символов. В нем реализованы в виде классов еще некоторые очень полезные типы. Наиболее интересные из них — **AnsiString**, имеющий множество методов и перегруженных операций, облегчающих работу со строками, и типы списков строк **TStrings** и **TStringList**. Первый из них рассмотрен в следующем разделе. А описание второго и третьего вы найдете в главе 16.

13.4.2 Тип строк **AnsiString**

В C++Builder тип строк **AnsiString** реализован как класс, объявленный в файле **vcl/dstring.h** и аналогичный типу длинных строк в Delphi. Это строки с нулевым символом в конце. При объявлении переменные типа **AnsiString** инициализируются пустыми строками.

Для **AnsiString** определены операции отношения **==**, **!=**, **>**, **<**, **>=**, **<=**. Сравнение производится с учетом регистра. Сравниваются коды символов, начиная с первого, и если очередные символы не одинаковы, строка, содержащая символ с меньшим кодом, считается меньше. Если все символы совпали, но одна строка длиннее и в ней имеются еще символы, то она считается больше, чем более короткая.

Для **AnsiString** определены операции присваивания **=**, **+=** и операция склеивания строк (конкатенации) **+**. Определена также операция индексации **[]**. Индексы начинаются с 1. Например, если **S1** = «Привет», то **S1[1]** вернет 'П', **S1[2]** вернет 'р' и т.д.

Класс **AnsiString** имеет множество методов, подробно рассмотренных в главе 16. Не останавливаясь сейчас на их перечислении, рассмотрим только некоторые примеры применения типа **AnsiString**.

Тип **AnsiString** используется для ряда свойств компонентов C++Builder. Например, для таких, как свойство **Text** окон редактирования, свойства **Caption** меток и разделов меню и т.д. Этот же тип используется для отображения отдельных строк в списках строк типа **TStrings**. Таким образом, постоянно имея дело с этими свойствами, вы постоянно работаете с **AnsiString**.

Рассмотрим некоторые примеры работы с **AnsiString**. Следующий оператор демонстрирует конкатенацию (склеивание) двух строк:

```
Label1->Caption = Edit1->Text + ' ' + Edit2->Text;
```

В данном случае в свойстве **Label1->Caption** отображается текст, введенный пользователем в окне редактирования **Edit1**, затем записывается символ пробела, а затем — текст, введенный в окне редактирования **Edit2**.

Как видите, склеивание строк типа **AnsiString** легко осуществляется перегруженной операцией сложения «+». Сравните это с теми вложенными вызовами функций **strcat**, которые приходилось делать в предыдущем разделе для тех же операций со строками типа (**char ***), и вы почувствуете преимущества **AnsiString**.

Теперь попробуем повторить рассмотренный в предыдущем разделе поиск в строке **S1** фрагмента, заданного строкой **S2**, и замену его текстом строки **S3**. Код, осуществляющий эти операции, может иметь вид:

```
AnsiString S1, S2, S3;  
// операторы занесения текста в S1, S2, S3  
...  
int i = S1.Pos(S2);  
if(i)  
    Label1->Caption = S1.SubString(1,i-1) + S3 +  
                      S1.SubString(i+S2.Length(),255);  
else Label1->Caption = "Текст не найден";
```

В этом коде использован ряд функций-элементов класса **AnsiString**: **Pos**, **SubString**, **Length**. Обратите внимание на то, что доступ к ним осуществляется операцией точка (**.**), вместо более привычной в C++Builder операции доступа к методам компонентов стрелка (**->**). Дело в том, что к методам компонентов доступ осуществляется через указатель на объект, а в данном случае к методам **AnsiString** доступ осуществляется через сами объекты — строки.

Первый выполняемый оператор приведенного кода использует функцию **Pos**. Эта функция ищет в строке, к которой она применена (в нашем случае в **S1**), первое вхождение подстроки, заданной ее параметром (в нашем случае **S2**). Если поиск успешный, функция возвращает индекс первого символа найденного вхождения подстроки. Индексы начинаются с 1. Если подстрока не найдена, возвращается 0.

Следующий оператор с помощью структуры **if...else** проверяет, не равно ли нулю (**false**) возвращенное функцией **Pos** значение. Если не равно, то производится формирование строки с заменой найденной подстроки. Строка формируется склеиванием трех строк: начальной части строки **S1**, расположенной до найденного вхождения подстроки, строки **S3**, заменяющей найденное вхождение, и заключительной части строки **S1**, расположенной после найденного вхождения. Для получения фрагментов строки **S1** использована функция **SubString**. Эта функция возвращает подстроку, начинающуюся с символа в позиции, заданной первым параметром функции, и содержащую число символов, не превышающее значение, заданное вторым параметром функции. Таким образом, выражение **S1.SubString(1, i - 1)** возвращает подстроку строки **S1**, начинающуюся с первого символа и содержащую **i - 1** символов, т.е. часть строки **S1**, расположенную до найденного вхождения подстроки **S2**. Аналогично, выражение **S1.SubString(i + S2.Length(), 255)** возвращает подстроку строки **S1**, расположенную после найденного вхождения подстроки **S2**. При этом для определения начала этой подстроки использована функция **Length**, возвращающая число символов в строке (в нашем случае — в строке **S2**, содержащей заменяемый фрагмент). В приведенном выражении в качестве второго параметра функции **SubString** задано число 255, которое, как ожидается, превышает длину подстроки. В действительности будет возвращено менее 255 символов, столько, сколько имеется до завершающего **S1** нулевого символа.

Сравнение данного кода с приведенным в предыдущем разделе для типов строк (**char ***), как мне кажется, показывает большую прозрачность действий со строками **AnsiString**.

Если нам надо не отображать измененную строку в виде сообщения, а просто произвести замену фрагмента в исходной строке **S1**, это еще более упрощает код, который в этом случае сводится всего к двум операторам:

```
int i = S1.Pos(S2);  
S1 = S1.SubString(1,i-1) + S3 + S1.SubString(i+S2.Length(),255);
```

Подобная задача для строк (**char ***) была бы более сложной и потребовала бы объявления дополнительного буфера для временного хранения формируемой строки.

Давайте еще более усложним задачу: пусть в строке **S1** надо заменить все вхождения **S2** на строку **S3**. Эту задачу можно было бы решить следующим кодом:

```
int i0 = 0, i = S1.Pos(S2);
while(i)
{
    S1 = S1.SubString(1, i + i0 - 1) + S3 +
        S1.SubString(i + i0 + S2.Length(), 255);
    i0 += i - 1 + S3.Length();
    i = S1.SubString(i0 + 1, 255).Pos(S2);
}
```

Приведенный код мало отличается от рассмотренного ранее и не содержит каких-то новых функций. Основные отличия заключаются в следующем. Во-первых, вводится переменная **i0** — индекс, предшествующий первому символу еще не обработанной части строки **S1**. Значение **i0** изменяется после обработки очередной части строки. Во-вторых, очередное вхождение строки **S2** в **S1** определяется не по всей строке **S1**, а только по ее еще не обработанной части : **S1.SubString(i0 + 1, 255)**.

Рассмотренную задачу контекстного поиска и замены в строке можно было бы решить иначе, воспользовавшись функциями **Delete** и **Insert** класса **AnsiString**. Функция **Delete** удаляет из строки, начиная с позиции, заданной первым параметром функции, число символов, заданное вторым параметром функции. Функция **Insert** вставляет в строку подстроку, заданную первым параметром функции, в позицию, заданную вторым параметром функции.

Применение этих функций позволяет выполнить контекстную замену с помощью, например, следующего кода:

```
int i0 = 1, i = S1.Pos(S2);
while(i > i0)
{
    S1.Delete(i, S2.Length()); // удаление вхождения S2
    S1.Insert(S3, i);          // вставка S3
    i0 = i + S3.Length();
    i = i0 - 1 + S1.SubString(i0, 255).Pos(S2);
}
```

Мы проиллюстрировали применение только малой части методов, имеющихся в классе **AnsiString**. Полный перечень этих методов вы найдете в соответствующем разделе главы 16. В заключение отметим только метод, позволяющий переходить от типа **AnsiString** к типу (**char ***). Несмотря на то, что применение **AnsiString** практически всегда удобнее (**char ***), такие переходы приходится делать при передаче параметров в некоторые функции, требующие тип параметра (**char ***). Чаще всего это связано с вызовом функций API Windows или функций C++Builder, инкапсулирующих такие функции. Например, на протяжении этой книги многократно использовалась функция **Application->MessageBox**, требующая в качестве двух своих первых параметров (сообщения и заголовка окна) тип (**char ***). Аналогичные преобразования требуются для функции **PlaySound** для передачи в нее имени файла и для многих других функций.

Преобразование строки **AnsiString** в строку (**char ***) осуществляется функцией **c_str()** без параметров, возвращающей строку с нулевым символом в конце, содержащую текст той строки **AnsiString**, к которой она применена. Например, если вы имеете строки **S1** и **S2** типа **AnsiString**, которые хотите передать в функцию **Application->MessageBox** в качестве сообщения и заголовка окна, то вызов **Application->MessageBox** может иметь вид:

```
Application->MessageBox(S1.c_str(), S2.c_str(), MB_OK);
```

Возможно и обратное преобразование строки (**char ***) в строку **AnsiString**. Для этого используется функция

```
AnsiString(char *S)
```

которая возвращает строку типа **AnsiString**, содержащую текст, записанный в строке **S**, являющейся аргументом функции.

13.5 Перечислимые типы

Перечислимые типы определяют упорядоченное множество идентификаторов, представляющих собой возможные значения переменных этого типа. Вводятся эти типы для того, чтобы сделать код более понятным. В частности, многие типы C++Builder являются перечислимыми, что упрощает работу с ними, поскольку дает возможность работать не с абстрактными числами, а с осмысленными значениями.

Приведем пример, который покажет смысл введения пользователем своего перечислимого типа. Пусть, например, в программе должна быть переменная **Mode**, в которой зафиксирован один из возможных режимов работы приложения: чтение данных, их редактирование, запись данных. Можно, конечно, дать переменной **Mode** тип **int** и присваивать этой переменной в нужные моменты времени одно из трех условных чисел: 0 — режим чтения, 1 — режим редактирования, 2 — режим записи. Тогда программа будет содержать операторы вида

```
if (Mode == 1) ...
```

Через некоторое время уже забудется, что означает значение **Mode**, равное 1, и разбираться в таком коде будет очень сложно. А можно поступить иначе: определить переменную **Mode** как переменную перечислимого типа и обозначить ее возможные значения как **mRead**, **mEdit**, **mWrite**. Тогда приведенный выше оператор изменится следующим образом:

```
if (Mode == mEdit) ...
```

Конечно, такой оператор понятнее, чем предыдущий.

Хороший стиль программирования

Не вводите в свои программы числовые константы, отображающие какие-то режимы работы или состояния объектов. Пользуйтесь для этого перечислимыми типами. Тогда код программы становится намного понятнее и легче осуществлять его модификацию и сопровождение.

Переменные перечислимого типа могут определяться предложением вида:

```
enum {<константа 1>, ..., <константа n>} <имена переменных>;
```

Например

```
enum {mRead, mEdit, mWrite} Mode;
```

Этот оператор вводит именованные константы **mRead**, **mEdit**, **mWrite** и переменную **Mode**, которая может принимать значения этих констант. В момент объявления переменная инициализируется значением первой константы, в нашем примере — **mRead**. В дальнейшем вы можете присваивать ей любые допустимые значения. Например:

```
Mode = mEdit;
```

Значение переменной перечислимого типа можно проверять, сравнивая ее величину с возможными значениями. Кроме того, надо учитывать, что перечислимые типы относятся к целым порядковым типам и к ним применимы любые операции сравнения: **>**, **<** и т.п. Например, вы можете писать операторы:

```
if (Mode > mRead) ...;  
if (Mode < mWrite) ...;  
if (Mode == mEdit) ...;
```

Вы можете также использовать **Mode** в структуре **switch**:

```
switch (Mode)
{
    case mRead:    ...
                  break;
    case mEdit:    ...
                  break;
    case mWrite:   ...
}
```

По умолчанию перечислимые значения, указанные в объявлении **enum**, интерпретируются как целые числа, причем первое значение эквивалентно 0, второе — 1 и т.д. Именно эти значения рассматриваются в операциях отношения **>**, **<** и др. Значения по умолчанию можно изменить, если после имени константы указать знак равенства (**=**) и задать присваиваемое целое значение, как положительное, так и отрицательное. Например:

```
enum {mRead = -1, mEdit, mWrite = 2} Mode;
```

Если после каких-то констант не задано их целое значение, оно считается на 1 больше предыдущего. Поэтому в приведенном примере **mRead** эквивалентно -1, **mEdit** эквивалентно 0, **mWrite** эквивалентно 2.

После ключевого слова **enum** может следовать тэг — имя объявляемого типа. Например:

```
enum regim {mRead = -1, mEdit, mWrite = 2} Mode, Model;
```

Этот оператор объявляет две переменные **Mode** и **Model** перечислимого типа, и кроме того определяет тип **regim**. В дальнейшем вы можете воспользоваться именем **regim** для объявления каких-то новых переменных, например:

```
regim Mode3;
```

13.6 Множества

Множество — это группа элементов, которая ассоциируется с ее именем и с которой можно сравнивать другие величины, чтобы определить, принадлежат ли они этому множеству. Как частный случай, множество может быть пустым.

Множество реализовано в C++Builder как шаблон класса, определенный в головном файле **vcl/sysdefs.h**.

Объявляется множество оператором:

```
Set <type, minval, maxval> переменные;
```

Параметр **type** определяет тип элементов множества. Обычно это порядковые типы **int**, **char** или перечислимый. Параметры **minval** и **maxval** типа **unsigned char** определяют минимальное и максимальное значения элементов множества. Минимальное значение должно быть не меньше 0, максимальное — не более 255.

Приведем примеры объявления множеств.

Объявление переменной **s1** как множества всех заглавных латинских букв имеет вид:

```
Set <char, 'A', 'Z'> s1;
```

Следующий оператор объявляет множество **Ch**, содержащее все символы:

```
Set <char, 0, 255> Ch;
```

Следующие операторы объявляют тип **UPPERCASESet** множества всех заглавных латинских букв и объявляют переменные **s2** и **s3** этого типа:

```
typedef Set <char, 'A', 'Z'> UPPERCASESet;
UPPERCASESet s1, s2;
```

Следующие операторы определяют множество **S**, элементами которого являются данные перечислимого типа **E**: **red, yellow, green**:

```
enum E { white, red, yellow, green };
Set <E, red, green> S;
```

Объявление переменной типа множества **Set** не инициализирует ее какими-то значениями. Инициализацию можно делать с помощью описанной ниже операции **<<** — добавление элемента в множество.

Для множества определены следующие операции (в описании операций словами «данное множество» обозначается левый операнд):

Операция	Определение	Описание
-	Set __fastcall operator -(const Set& rhs) const;	данное множество равно разности двух множеств: данного и rhs (операция xor с их элементами)
-=	Set& __fastcall operator -=(const Set& rhs);	создание нового множества, определенного разностью двух множеств: данного и rhs (операция xor с их элементами)
*	Set& __fastcall operator *=(const Set& rhs);	создание нового множества, определенного пересечением двух множеств: данного и rhs (операция and с их элементами)
*=	Set __fastcall operator *(const Set& rhs) const;	данное множество равно пересечению двух множеств: данного и rhs (операция and с их элементами)
+	Set __fastcall operator +(const Set& rhs) const;	создание нового множества, определенного объединением двух множеств: данного и rhs (операция or с их элементами)
+=	Set& __fastcall operator +=(const Set& rhs);	данное множество равно объединению двух множеств: данного и rhs (операция or с их элементами)
<<	Set& __fastcall operator <<(const T el);	добавление элемента el в данное множество
<<	friend ostream& operator <<(ostream& os, const Set& arg);	поместить множество arg в поток ostream (выводится 0 или 1 для каждого элемента в зависимости от его наличия в множестве)

Операция	Определение	Описание
>>	Set& __fastcall operator >>(const T el);	удаление элемента el из данного множества
>>	friend istream& operator >>(istream& is, Set& arg);	извлечь множество arg из потока istream (вводится 0 или 1 для каждого элемента в зависимости от его наличия в множестве)
=	Set& __fastcall operator =(const Set& rhs);	присваивание данному множеству содержимого множества rhs
==	bool __fastcall operator ==(const Set& rhs) const;	эквивалентность двух множеств: данного и rhs (совпадение всех элементов)
!=	bool __fastcall operator !=(const Set& rhs) const ;	неэквивалентность двух множеств: данного и rhs

Все операции можно применять только к множествам одного типа, то есть к таким, при объявлении которых все аргументы объявления (**type**, **minval** и **maxval**) совпадают. В операциях, создающих новое множество (операции **+**, **-** и *****), переменная, в которую заносится результат, также должна быть того же типа, что и операнды. Операция эквивалентности возвращает **true** в случае, когда оба операнда содержат только совпадающие элементы. Соответственно только в этом случае операция неэквивалентности возвращает **false**.

Для множеств **Set** определены также два метода:

Метод	Определение	Описание
Clear	Set& __fastcall Clear();	очистка множества
Contains	bool __fastcall Contains(const T el) const;	проверка наличия в множестве элемента el

Рассмотрим примеры работы с множествами. Пусть вы задаете пользователю в программе некоторый вопрос, подразумевающий ответ типа «Yes/No». Тогда возможные символы, вводимые пользователем в качестве ответа, являются множеством, содержащим символы «y», «Y», «n» и «N». Сформировать такое множество можно операторами:

```
Set <char, 0, 255> TrueKey;
...
TrueKey << 'y' << 'Y' << 'n' << 'N';
```

Тогда проверить, принадлежит ли введенный пользователем символ **Key** множеству допустимых ответов, можно с помощью метода **Contains**:

```
if (!TrueKey.Contains(Key))
    ShowMessage("Вы ввели ошибочный ответ");
else ...
```

Рассмотрим еще один пример. Пусть вы хотите, чтобы в окне редактирования **Edit1** пользователь мог вводить только число, т.е. только цифры от 0 до 9. Это можно сделать, включив в обработчик события **OnKeyPress** этого окна операторы:

```
Set <char, '0', '9'> Dig;
Dig << '0' << '1' << '2' << '3' << '4' << '5'
  << '6' << '7' << '8' << '9';
if (!Dig.Contains(Key))
  {Key = 0; Beep();}
```

При попытке пользователя ввести символ, отличный от цифры, раздастся звук (его обеспечит функция **Beep**) и символ не появится в окне. Подробнее этот пример рассмотрен в главе 4 в разделе 4.3.2.2.

13.7 Указатели

Указатель — это переменная, значение которой равно значению адреса памяти, по которому лежит значение некоторой другой переменной. В этом смысле имя этой другой переменной отправляет к ее значению *прямо*, а указатель — *косвенно*. Ссылка на значение посредством указателя называется *косвенной адресацией*.

Указатели, подобно любым другим переменным, перед своим использованием должны быть объявлены. Объявление указателя имеет вид:

```
type *ptr;
```

где **type** — один из предопределенных или определенных пользователем типов, а **ptr** — указатель. Читается это объявление так: «**ptr** является указателем на значение типа **type**».

Например,

```
int *countPtr, count;
```

объявляет переменную **countPtr** типа **int *** (т.е. указатель на целое число) и переменную **count** целого типа. Символ ***** в объявлении относится только к **countPtr**. Каждая переменная, объявляемая как указатель, должна иметь перед собой знак звездочки (*). Если в приведенном примере желательно, чтобы и переменная **count** была указателем, надо записать:

```
int *countPtr, *count;
```

Символ ***** в этих записях обозначает *операцию косвенной адресации*.

Может быть объявлен и указатель на **void**:

```
void *Pv;
```

Это универсальный указатель на любой тип данных. Но прежде, чем его использовать, ему надо в процессе работы присвоить значение указателя на какой-то конкретный тип данных. Например:

```
Pv = countPtr;
```

Хороший стиль программирования

Хотя это и не обязательно, включайте буквы **Ptr** или просто **P** в имена переменных указателей, чтобы и вам, и тем, кто будет сопровождать вашу программу, было ясно, что эти переменные являются указателями и требуют соответствующей обработки.

В **C++Builder** указатели используются очень широко. В частности, все компоненты, формы и т.д. объявляются именно как указатели на соответствующий объект. Посмотрев заголовочный файл любого приложения, вы увидите в нем объявления вида:

```
TForm1 *Form1;
TButton *Button1;
```

Указатели должны инициализироваться либо при своем объявлении, либо с помощью оператора присваивания. Указатель может получить в качестве началь-

ного значения 0, **NULL** или адрес. Указатель с начальным значением 0 или **NULL** ни на что не указывает. **NULL** — это символическая константа, определенная специально для цели показать, что данный указатель ни на что не указывает. Пример объявления указателя с его инициализацией:

```
int *countPtr = NULL;
```

Хороший стиль программирования

Присваивайте указателям при их объявлении значение **NULL** или адрес. Неинициализированный указатель может при работе с ним приводить к самым неожиданным результатам.

Для присваивания указателю адреса некоторой переменной используется *операция адресации* **&**, которая возвращает адрес своего операнда. Например, если имеются объявления

```
int y = 5;
int *yPtr, x;
```

то оператор

```
yPtr = &y;
```

присваивает адрес переменной **y** указателю **yPtr**.

Для того, чтобы получить значение, на которое указывает указатель, используется операция *****, обычно называемая *операцией косвенной адресации* или *операцией разыменования*. Она возвращает значение объекта, на который указывает ее операнд (т.е. указатель). Например, если продолжить приведенный выше пример, то оператор

```
x = *yPtr
```

присвоит переменной **x** значение 5, т.е. значение переменной **y**, на которую указывает **yPtr**.

Предупреждение

Операцию разыменования нельзя применять к указателю на **void**, поскольку для него неизвестно, какой размер памяти надо разыменовывать.

Массивы и указатели в Си++ тесно связаны и могут быть использованы почти эквивалентно. Имя массива можно понимать как константный указатель на первый элемент массива. Его отличие от обычного указателя только в том, что его нельзя модифицировать.

Указатели можно использовать для выполнения любой операции, включая индексирование массива. Пусть вы сделали следующее объявление:

```
int b[5] = {1, 2, 3, 4, 5}, *Pt;
```

Тем самым вы объявили массив целых чисел **b[5]** и указатель на целое **Pt**. Поскольку имя массива является указателем на первый элемент массива, вы можете задать указателю **Pt** адрес первого элемента массива **b** с помощью оператора

```
Pt = b;
```

Это эквивалентно присваиванию адреса первого элемента массива следующим образом

```
Pt = &b[0];
```

Теперь можно сослаться на элемент массива **b[3]** с помощью выражения ***(Pt + 3)**.

Указатели можно индексировать точно так же, как и массивы. Например, выражение **Pt[3]** ссылается на элемент массива **b[3]**.

Таким образом манипуляции, определенные для массивов, определены и для указателей на массивы. Но с точки зрения понятности программы лучше это без крайней необходимости не использовать.

Указатели могут применяться как операнды в арифметических выражениях, выражениях присваивания и выражениях сравнения. Однако, не все операции, обычно используемые в этих выражениях, разрешены применительно к переменным указателям.

С указателями может выполняться ограниченное количество арифметических операций. Указатель можно увеличивать (++), уменьшать (--), складывать с указателем целые числа (+ или +=), вычитать из него целые числа (- или -=) или вычитать один указатель из другого.

Сложение указателей с целыми числами отличается от обычной арифметики. Прибавить к указателю 1 означает сдвинуть его на число байтов, содержащихся в переменной, на которую он указывал. Обычно подобные операции применяются к указателям на массивы. Если продолжить приведенный выше пример, в котором указателю **Pt** было присвоено значение **b** — указателя на первый элемент массива, то после выполнения оператора

```
Pt += 2;
```

Pt будет указывать на третий элемент массива **b**. Истинное же значение указателя **Pt** изменится на число байтов, занимаемых одним элементом массива, умноженное на 2. Например, если каждый элемент массива **b** занимает 2 байта, то значение **Pt** (т.е. адрес в памяти, на который указывает **Pt**) увеличится на 4.

Аналогичные правила действуют и при вычитании из указателя целого значения.

Переменные указатели можно вычитать один из другого. Например, если **Pt** указывает на первый элемент массива **b**, а указатель **Pt1** — на третий, то результат выражения **Pt1 - Pt** будет равен 2 — разности индексов элементов, на которые указывают эти указатели. И так будет, несмотря на то, что адреса, содержащиеся в этих указателях, различаются на 4 (если элемент массива занимает 2 байта).

Арифметика указателей теряет всякий смысл, если она выполняется не над указателями на массив. Нельзя полагать, чтоб две переменные одинакового типа хранятся в памяти плотную друг к другу, если только они не соседствуют в массиве. Сравнение указателей операциями **>**, **<**, **>=**, **<=** также имеют смысл только для указателей на один и тот же массив. Однако, операции отношения **==** и **!=** имеют смысл для любых указателей. При этом указатели равны, если они указывают на один и тот же адрес в памяти.

Указатель можно присваивать другому указателю, если оба указателя имеют одинаковый тип. В противном случае нужно использовать операцию приведения типа, чтобы преобразовать значение указателя в правой части присваивания к типу указателя в левой части присваивания. Исключением из этого правила является указатель на **void** (т.е. **void***), который является общим указателем, способным представлять указатели любого типа. Указателю на **void** можно присваивать все типы указателей без приведения типа. Однако указатель на **void** не может быть присвоен непосредственно указателю другого типа — указатель на **void** сначала должен быть приведен к типу соответствующего указателя.

Мы рассмотрели ранее указатели на массивы. Однако, соотношение между массивами и указателями может быть и обратным — могут использоваться массивы указателей. Подобные структуры часто используются в массивах строк или в массивах указателей на различные объекты.

Например, вы можете сделать следующее объявление:

```
char *Sa[2] = {"Это первая строка", "Вторая"};
```

Вы объявили массив размером 2 элементов типа (**char ***). Каждый элемент такого массива — строка. Но в C++ строка является, по существу, указателем на ее первый символ. Таким образом, каждый элемент в массиве строк в действительности является указателем на первый символ строки. Каждая строка хранится в памяти как строка, завершающаяся нулевым символом. Число символов в каждой

из строк может быть различным. Таким образом, массив указателей на строки позволяет обеспечить доступ к строкам символов любой длины.

Указатели широко используются при передаче параметров в функции. Особенности использования указателей в этих целях см. в разделе 12.5.2 главы 12.

В разделе 1.5.6 главы 1 подробно рассмотрены указатели на объекты и работа с ними.

13.8 Ссылки

Ссылки — это специальный тип указателя, который позволяет работать с указателем как с объектом. Объявление ссылки делается с помощью операции ссылки, обозначаемой амперсантом (&) — тем же символом, который используется для адресации. Если в вашей программе имеется указатель на объект какого-то типа **MyObject**:

```
MyObject *P = new MyObject;
```

то вы можете создать ссылку на этот объект оператором:

```
MyObject & Ref = *P;
```

Объявленная таким образом переменная **Ref** является ссылкой на объект **MyObject**. Она может рассматриваться как псевдоним объекта. Эта переменная реально является указателем, а не самим объектом. Но работа с ней производится как с объектом. Например, если вы хотите получить доступ к некоторому свойству объекта **x**, то через указатель на объект вы обеспечиваете доступ выражением **P->x**, т.е. через операцию стрелка. А через ссылку вы обеспечиваете доступ к свойству **x** выражением **Ref.x**, т.е. через операцию точка.

Аналогичным образом вы можете получить доступ по ссылке и к любым компонентам. Например, если в вашем приложении имеется метка **Label1**, то вы можете обращаться к его свойству **Caption** оператором

```
Label1->Caption = "Это обращение по указателю";
```

А можете ввести соответствующую ссылку и обращаться через нее:

```
TLabel & ref = *Label1;  
ref.Caption = "Это обращение по ссылке";
```

Чаще всего ссылки используются при передаче в функции параметров по ссылке. Этот вопрос подробно рассмотрен в главе 12 в разделе 12.5.2.

13.9 Файлы и потоки

Работа с файлами в C++Builder может производиться несколькими принципиально различными (с точки зрения пользователя) способами:

- использование библиотечных компонентов
- работа с файлами как с потоками в стиле C
- работа с файлами как с потоками в стиле C++

Рассмотрим эти возможности.

13.9.1 Файловый ввод/вывод с помощью компонентов

Работа с текстовыми файлами может осуществляться с помощью методов **LoadFromFile** и **SaveToFile**, имеющихся у классов **TStrings** и **TStringList**. Эти классы описывают списки строк и обладают множеством методов, позволяющих манипулировать строками.

Если вы хотите в своем приложении прочитать содержимое некоторого текстового файла, обработать текст и сохранить его в файле, вы можете сделать это следующим образом. Объявите и создайте две глобальные переменные: список типа **TStringList**, в котором будет храниться текст файла, и строковую переменную типа **AnsiString**, в которой можете сформировать имя файла. Например:

```
TStringList *List = new TStringList;  
AnsiString SFile = "Test.txt";
```

Не забудьте только, что если требуемый файл расположен не в текущем каталоге и вам надо указать путь к файлу, то обратные слэши в записи пути должны быть удвоенные (см. раздел 12.3.1 главы 12). Например, если вам требуется файл «с:\MyTest\Test.txt», то вы должны записать его как «с:\\MyTest\\Test.txt».

В момент, когда вы хотите загрузить в свой список файл, надо выполнить оператор

```
List->LoadFromFile(SFile);
```

Впрочем, ограничиться таким оператором можно, если есть уверенность, что требуемый файл существует. В противном случае код надо несколько усложнить, чтобы можно было перехватить сгенерированное исключение. Например:

```
try{  
    List->LoadFromFile(SFile);  
}  
catch(...){  
    ShowMessage("Файл \"" + SFile + "\" не найден");  
}
```

Если файл нормально загрузился в список **List**, вы можете работать с его текстом. Текст расположен в свойстве списка **Strings[int Index]**, в котором каждая строка имеет тип **AnsiString**. Индексы начинаются с нуля. Для нашего примера **List->Strings[0]** — это первая строка, **List->Strings[1]** — вторая и т.д.

Для списков типа **TStringList** предусмотрено множество методов, которые вы можете посмотреть в соответствующем разделе главы 16. При обработке отдельных строк вы можете использовать операции и методы, предусмотренные для строк типа **AnsiString** (см. раздел 13.4.2 и соответствующий раздел главы 16).

Если вы хотите сохранить файл после проведенного редактирования, можно выполнить оператор

```
List->SaveToFile(SFile);
```

где **SFile** содержит прежнее или новое имя файла.

Если при открытии и сохранении файла вы хотите воспользоваться стандартными диалогами Windows, вы можете посмотреть раздел 3.8.2 главы 3, в котором подробно расписаны необходимые для этого действия.

Если вы открываете файл для того, чтобы пользователь мог его просмотреть, что-то в нем отредактировать и сохранить, вы можете обойтись без описанного выше объекта типа **TStringList**. Для этих целей проще воспользоваться многострочными окнами редактирования типов **TMemo** или **TRichEdit**. В последнем случае вы можете работать не только с обычными текстовыми файлами, но и с файлами в обогащенном формате RTF. Свойства **Lines** этих компонентов имеют тип **TStrings**, что позволяет применять к ним непосредственно методы **LoadFromFile** и **SaveToFile**. Например:

```
Memol->Lines->LoadFromFile(SFile);  
RichEdit1->Lines->LoadFromFile(SFile);
```

Работа с компонентами **Memo** и **RichEdit** подробно рассмотрена в главе 3 в разделе 3.2.4.

Через компоненты **C++Builder** можно работать не только с текстовыми файлами, но и с файлами изображений и мультимедиа. Этим вопросам посвящена глава 5.

13.9.2 Файловый ввод/вывод с помощью потоков в стиле C

13.9.2.1 Общие сведения

В языках C и C++ файл рассматривается как поток (stream), представляющий собой последовательность считываемых или записываемых байтов. При этом поток «не знает», что и в какой последовательности в него записано. Расшифровка смысла записанных последовательностей байтов лежит на программе.

Классический подход, принятый в C, заключается в том, что информация о потоке (файле) заносится в структуру типа **FILE**, определенную в файле **stdio.h**. Файл открывается с помощью функции **fopen**, которая возвращает указатель на структуру типа **FILE**. Этот указатель потока используется далее во всех операциях с файлами.

Синтаксис функции **fopen**:

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Функция **fopen** открывает файл с именем в виде строки, на которую ссылается указатель **filename**, и связывает с ним поток. Аргумент **mode** указывает на строку, которая определяет режим открытия. Она может содержать спецификаторы:

r	открыть файл только для чтения
r+	открыть существующий файл для чтения и записи
a	открыть или создать файл для записи данных в конец файла
a+	открыть или создать файл для чтения или записи в конец файла
w	создать файл для записи
w+	создать файл для чтения и записи

К указанным спецификаторам в конце или перед символом «+» может добавляться символ «**t**» — текстовый файл, или «**b**» — бинарный, двоичный файл. Например, **rt**, **rb**, **r+t**, **r+b** и т.д. Если ни символ «**t**», ни символ «**b**» не указаны, то тип открываемого файла определяется значением глобальной переменной **_fmode**, определенной в файле **fcntl.h**. Она может принимать значения **O_TEXT** — текстовый файл (по умолчанию) или **O_BINARY** — двоичный файл. Более подробное пояснение режимов открытия файлов вы найдете в главе 15 в разделе 15.5.2.

Открываемый функцией **fopen** поток буферизуется, т.е. обмен информацией происходит не непосредственно с файлом, а с промежуточным буфером, расположенным в оперативной памяти. Информация переписывается из буфера в файл только при переполнении буфера или при закрытии файла. В главе 15 в разделе 15.5.2 вы можете посмотреть функции, управляющие процессом буферизации.

Функция **fopen** возвращает указатель на объект, управляющий потоком. Если попытка открыть файл закончилась неудачей, **fopen** возвращает нулевой указатель.

После того, как необходимая работа с файлом (чтение или запись) завершена, файл должен быть закрыт функцией **fclose(FILE *)**, в которую передается указатель потока.

13.9.2.2 Текстовые файлы

Рассмотрим сначала работу с текстовыми файлами. Открытие текстового файла «Test.txt» может иметь вид:

```
#include <stdio.h>
...
FILE *F;
```

```

if ((F = fopen("Test.txt", "rt")) == NULL)
{
    ShowMessage("Файл не удается открыть");
    return;
}
...           // чтение из файла
fclose(F);    // закрытие файла

```

Здесь объявляется переменная **F** — указатель потока и связывается с файлом «Test.txt», открываемым как текстовый только для чтения. Если открыть файл не удалось (например, он не существует), появляется сообщение об ошибке.

Из открытого таким образом файла можно читать информацию. После окончания чтения файл должен быть закрыт функцией **fclose(F)**.

Если бы файл открывался функцией

```
fopen("Test.txt", "rt+")
```

то из такого файла можно было бы не только читать информацию, но и записывать в него новые строки.

Из текстового файла можно читать информацию по строкам или по символам. Чтение строки осуществляется функцией **fgets**:

```
char *fgets(char *s, int n, FILE *stream);
```

В вызове функции **s** — указатель на буфер, в который читается строка, **n** — число читаемых символов. Чтение символов в строку происходит или до появления символа конца строки «\n» (этот символ записывается в строку), или читается **n-1** символ. В конце прочитанной строки записывается нулевой символ.

Например, чтение и отображение в компоненте **Memo1** всех строк файла может быть организовано следующим образом:

```

char s[80];
Memo1->Clear();
do
{
    fgets(s, 80, F);
    if (feof(F)) break;
    if (s[strlen(s)-1] == '\n') s[strlen(s)-1] = 0;
    Memo1->Lines->Add(s);
}
fclose(F); // закрытие файла

```

Функция **fgets** читает очередную строку. Функция **feof** проверяет, не прочитан ли символ конца файла. При чтении этого символа **feof** возвращает ненулевое значение и цикл прерывается. Если признака конца файла нет, то оператор

```
if (s[strlen(s)-1] == '\n') s[strlen(s)-1] = 0;
```

убирает из строки последний символ, если он оказывается символом перевода строки. Эта операция не обязательна, но наличие символа «\n» испортит вид строки в окне **Memo1**. Затем прочитанная строка заносится в окно редактирования.

Чтение из текстового файла форматированных данных может осуществляться функцией **fscanf**.

```
int fscanf(FILE *stream, const char *format[, address, ...]);
```

Ее параметр **format** определяет строку форматирования аргументов, заданных своими адресами. Подробно строка форматирования рассмотрена в главе 15 в разделе 15.1.4.2. Сейчас отметим только, что эта строка при чтении обычно состоит из последовательности символов «%», после которых следует символ типа читаемых данных. Ниже приведены некоторые наиболее часто используемые символы (подробнее см. в разделе 15.1.4.2 главы 15):

Символ	Вводимое значение	Тип аргумента функции
i	Десятичное, восьмеричное или шестнадцатеричное целое	int *arg
l	Десятичное, восьмеричное или шестнадцатеричное целое	long *arg
d	Десятичное целое	int *arg
D	Десятичное целое	long *arg
u	Десятичное целое без знака	unsigned int *arg
U	Десятичное целое без знака	unsigned long *arg
e, E	Действительное с плавающей запятой	float *arg
s	Строка символов	char arg[]
c	Символ	char *arg

Перед символом типа могут добавляться модификаторы. В частности, модификатор **l** расширяет тип целого до **long int**, а тип действительного до **double**.

Пусть, например, вы знаете, что начиная с текущей позиции файла в нем записаны, разделенные пробелами, два целых и одно действительное число. Тогда прочитать эти числа можно операторами:

```
int i1, i2;
double r;
fscanf(F, "%d%d%le", &i1, &i2, &r);
```

Предупреждение

Обратите внимание, что в качестве аргументов, в которые заносятся читаемые функцией **fscanf** данные, всегда указываются адреса переменных, а не сами переменные. Отсутствие операции адресации (&) — очень распространенная ошибка, которая приводит к самым неожиданным результатам.

При форматированном чтении могут возникать ошибки из-за достижения конца файла или из-за неверного формата записанного в файле числа. Проверить, успешно ли прошло чтение, можно по значению, возвращаемому функцией **fscanf**. При успешном чтении она возвращает число прочитанных полей. Поэтому в нашем примере лучше организовать чтение следующим образом:

```
if (fscanf(F, "%d%d%le", &i1, &i2, &r) != 3)
{
    ShowMessage("Ошибка чтения");
    ...
}
```

Символ типа **s** позволяет читать из файла отдельное слово, точнее — так называемую лексему — последовательность символов, завершающуюся пробельным символом. Пусть, например, мы хотим просмотреть файл, чтобы узнать, не встречается ли в нем слово, которое пользователь ввел в окне редактирования **Edit1**. Для решения этой задачи после того, как файл открыт, можно выполнить, например, следующий код:

```
char s[80], key[10];
strcpy(key, Edit1->Text.c_str()); // загрузка ключевой строки
do
{
    fscanf(F, "%s", &s);
```



```
if (feof(F) || !strcmp(s, key)) break;
}
while (true);
fclose(F);
if (!strcmp(s, key))
    ShowMessage("Слово найдено");
```

В этом коде вводится рабочая строка **s** и строка **key**, в которую функцией **strcpy** загружается ключевое слово. Далее в цикле функцией **fscanf** в строку **s** читается по одной лексеме из файла. Функция **strcmp** сравнивает эту лексему с ключом. Она возвращает 0, если строки **s** и **key** совпадают. В этом случае, а также при достижении конца файла цикл прерывается.

Мы рассмотрели вопросы чтения из текстового файла. Имеется также ряд функций записи в текстовый файл. Наиболее часто используемая из них — функция **fprintf**:

```
int fprintf(FILE *stream, const char *format[, argument, ...]);
```

Эта функция подобна рассмотренной выше функции **fscanf**, только строка форматирования строится несколько иначе. В ней используются аналогичные рассмотренным ранее символы типа, помещаемые после символа «%». Но имеется много возможностей по выбору формата печати данных в файл. Кроме того, все символы строки форматирования, не предваряемые символом «%», просто помещаются в выходной поток. Подробнее о функции **fprintf** и других функциях записи вы можете посмотреть в главе 15 в разделе 15.5.4. А пока приведем только короткий пример.

Пусть у вас имеется строка **s** типа (**char ***), содержащая фамилию сотрудника, и целое число **year**, содержащее год его рождения. Вы хотите создать текстовый файл и занести в него запись, первая строка которой содержит слово «ХАРАКТЕРИСТИКА», а вторая — текст «сотрудника ..., ... г.р.». Вместо точек в этом тексте подразумевается фамилия и год рождения. Это можно сделать следующим кодом:

```
FILE *F;
if ((F = fopen("Test.txt", "wt")) == NULL)
{
    ShowMessage("Файл не удастся создать");
    return;
}
char S[40];
int year = 1960;
strcpy(s, "Иванов");
fprintf(F, "ХАРАКТЕРИСТИКА\nсотрудник %s, %i г.р.\n", &S,
        year);
fclose(F);
```

Файл открывается функцией **fopen** как текстовый файл для записи. Если файла с указанным именем не было, он создается. Если такой файл был, все его содержимое уничтожается. Затем функцией **fprintf** в файл записывается требуемый текст. В строке форматирования этой функции записан текст первой строки, затем указан символ перехода на новую строку «\n»; далее следует начало второй строки — «сотрудник», затем символы «%s», задающие тип первого аргумента — указателя на **S**, затем символы «%i», задающие тип второго аргумента — числа **year**, и наконец — заключительная часть второй строки. В результате в файл будут записаны строки:

```
ХАРАКТЕРИСТИКА
сотрудник Иванов, 1960 г.р.
```

Рассмотренными функциями не ограничиваются возможности работы с текстовыми файлами. Подробнее все эти функции вы можете посмотреть в главе 15 в разделе 15.5.4.

13.9.2.3 Двоичные файлы

Теперь остановимся коротко на работе с двоичными файлами. Двоичный файл представляет собой просто последовательность символов, в которой без каких-либо разделителей — пробелов, символов конца строки и т.п. хранятся символы, отображающие самые различные объекты. Они совпадают с тем, как хранятся соответствующие объекты в оперативной памяти. Что именно и в какой последовательности лежит в двоичном файле — должна знать программа.

Двоичные файлы имеют немало преимуществ перед текстовыми при хранении каких-то числовых данных. Операции чтения и записи с такими файлами производятся намного быстрее, чем с текстовыми, поскольку отсутствует необходимость форматирования: перевода в текстовое представление и обратно. Двоичные файлы, как правило, имеют существенно меньший объем, чем аналогичные текстовые файлы. В двоичных файлах вы можете перемещаться в любую позицию и читать или записывать данные в произвольной последовательности, в то время как в текстовых файлах практически всегда производится последовательная обработка информации. Пожалуй, недостаток двоичного файла с точки зрения программиста только один — просматривая его с помощью какого-то текстового редактора, трудно понять, где что в нем находится, и это в ряде случаев затрудняет отладку.

О том, как открываются двоичные файлы, уже рассказывалось в разделе 13.9.2.1. Запись и чтение в двоичные файлы чаще всего производятся соответствующими функциями **fwrite** и **fread**:

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

В обе функции передается указатель **ptr** на выводимые или вводимые данные. Параметр **size** задает размер в байтах передаваемых данных, а параметр **n** определяет число передаваемых данных. Применение этих функций иллюстрируется приведенным ниже примером.

```
int i = 1, j = 25, il, jl;
double a = 25e6, al;
char s[10], sl[10];
strcpy(s, "Иванов");

FILE *F;
// запись в файл
if ((F = fopen("Test.dat", "wb")) == NULL)
{
    ShowMessage("Файл не удастся создать");
    return;
}
fwrite(&i, sizeof(int), 1, F);           // запись i
fwrite(&j, sizeof(int), 1, F);           // запись j
fwrite(&a, sizeof(double), 1, F);        // запись a
fwrite(s, sizeof(char), strlen(s)+1, F); // запись строки s
fclose(F);

// чтение из файла
if ((F = fopen("Test.dat", "rb")) == NULL)
{
    ShowMessage("Файл не удастся открыть");
    return;
}
fread(&il, sizeof(int), 1, F);           // чтение i
fread(&jl, sizeof(int), 1, F);           // чтение j
fread(&al, sizeof(double), 1, F);        // чтение a
fread(sl, sizeof(char), strlen(s)+1, F); // чтение строки s
fclose(F);
```

В данном примере создается двоичный файл «Test.dat» и в него записывается два целых числа *i* и *j*, действительное число *a* и строка *s*. Затем этот файл закрывается, открывается для чтения и данные из него читаются в переменные *il*, *jl*, *al* и *sl*.

В отношении записи и чтения чисел, вероятно, все понятно. А вопрос записи и чтения строк имеет смысл обсудить подробнее.

В приведенном примере запись строки производится оператором

```
fwrite(s, sizeof(char), strlen(s)+1, F); // запись строки s
```

Запись ведется по символам и указано число записываемых символов — **strlen(s)+1** (единица добавляется на нулевой символ в конце). Читается строка аналогично:

```
fread(sl, sizeof(char), strlen(s)+1, F); // чтение строки s
```

При этом чтение тоже идет по символам и читается **strlen(s)+1** символов.

Тут внимательный читатель может увидеть некоторую подтасовку. В данном учебном примере мы знаем длину строки, которую записали в файл, и можем прочитать требуемое число символов. Но как быть в реальных задачах, когда мы, скорее всего, не будем знать длину записанной строки? Эту проблему можно решить несколькими путями. Проще всего записывать и читать весь массив символов как единое целое:

```
fwrite(s, sizeof(s), 1, F);  
fread(sl, sizeof(s), 1, F);
```

Этот путь простой, но имеет один недостаток: записывается всегда весь массив символов *s*, даже если содержащаяся в нем строка много короче размера массива. Приведенные операторы в нашем примере эквивалентны операторам

```
fwrite(s, sizeof(char)*10, 1, F);  
fread(sl, sizeof(char)*10, 1, F);
```

Таким образом, при частичном заполнении массива в файле будут храниться лишние байты. Если в файле много строк разной длины, а все они будут храниться как максимальная из них, то размер файла будет значительно больше действительно необходимого.

Другой путь — записывать по-прежнему по символам, но при чтении проверять каждый символ, чтобы при появлении нулевого символа закончить чтение строки. Это может быть реализовано следующим образом:

```
// запись строки  
fwrite(s, sizeof(char), strlen(s)+1, F);  
...  
// чтение строки  
for(int ind = 0; ind < 10; ind++)  
{  
    fread(sl+ind, sizeof(char), 1, F);  
    if(sl[ind] == '\0') break;  
}
```

Здесь в цикле **for** читается за раз по одному символу и при обнаружении нулевого символа цикл прерывается. Обратите внимание, что адрес чтения очередного символа в данном случае задается выражением **sl+ind**. Нельзя было бы вместо этого использовать выражение **sl[ind]**, так как функция **fread** требует указания именно адреса, а не значения переменной, в которую осуществляется чтение.

Функцию **fread** в этом примере можно было бы заменить на **fgetc**, которая читает один символ из потока:

```
sl[ind] = fgetc(F);
```

И, наконец, еще один вариант чтения строк неизвестной длины из двоичного файла. Можно перед строкой записывать в файл целое число, равное числу символов в строке. Тогда чтение строки не встретит затруднений:

```
// запись строки
int it = strlen(s)+1;
fwrite(&it, sizeof(int), 1, F);
fwrite(s, sizeof(char), it, F);
...
// чтение строки
fread(&it, sizeof(int), 1, F);
fread(s1, sizeof(char), it, F);
```

В приведенных примерах чтение происходило последовательно. Но, работая с двоичными файлами, можно организовать произвольное чтение данных. Для этого служит указатель (курсор) файла, который определяет текущую позицию в файле для чтения и записи. При чтении или записи указатель автоматически смещается на число обработанных байтов. Узнать позицию указателя можно функцией **ftell**, которая возвращает текущую позицию:

```
long int ftell(FILE *stream);
```

Изменить позицию указателя можно функцией **fseek**:

```
int fseek(FILE *stream, long offset, int whence);
```

Эта функция задает сдвиг на число байтов **offset** относительно точки отсчета, определяемой параметром **whence**. Параметр **whence** может принимать значения:

Константа	whence	Точка отсчета
SEEK_SET	0	Начало файла
SEEK_CUR	1	Текущая позиция
SEEK_END	2	Конец файла

Если задано значение **whence = 1**, то **offset** может быть положительным (сдвиг вперед) или отрицательным (сдвиг назад).

Функция **rewind** перемещает указатель на начало файла (позиция 0). Впрочем, то же самое можно сделать оператором

```
fseek(F, 0L, 0);
```

Возможность перемещать указатель особенно полезна в файлах, которые состоят из однородных записей одинакового размера. Например, если в файле записаны только действительные числа типа **double**, то для того, чтобы прочитать *i*-ое число, достаточно выполнить операторы

```
fseek(F, sizeof(double)*(i-1), 0);
fread(&a, sizeof(double), 1, F);
```

Таким образом можно читать любые записи в любой последовательности.

С помощью перемещения указателя можно редактировать записи в файле. Пусть, например, вы хотите одно из чисел, записанных в файле, изменить, умножив его на 10. Это можно сделать, если открыть файл в режиме чтения и записи (например, «**rb+**»), установить позицию, соответствующую изменяемому числу, и выполнить операторы:

```
fread(&a, sizeof(double), 1, F);
a *= 10;
fseek(F, -sizeof(double), 1);
fwrite(&a, sizeof(double), 1, F);
```

Первый из этих операторов читает число в переменную **a**, второй — умножает его на 10. Третий оператор возвращает текущую позицию на одну запись назад, поскольку после выполнения **fread** позиция сдвинулась вперед. Последний оператор пишет в ту позицию, в которой было прочитано число, новое значение.

Ту же задачу можно решить иначе:

```
long int pos = ftell(F);           // запоминание позиции
fread(&a, sizeof(double), 1, F);
a *= 10;
fseek(F, pos, 0);                 // восстановление позиции
fwrite(&a, sizeof(double), 1, F);
```

Здесь функция **ftell** запоминает позицию, из которой читается число, а функция **fseek** восстанавливает эту позицию перед записью измененного числа.

С помощью двоичных файлов можно записывать и читать не только числа и строки, но и гораздо более сложные объекты, например, структуры. Ниже приведен пример, в котором определяется тип структуры **spers**, объявляются переменные этого типа **pers** и **pers1**, поля структуры **pers** заполняются, а затем она целиком записывается в файл. После того, как файл закрывается, он открывается для чтения и данные из него читаются в структуру **pers1**.

```
struct spers
{
    char Name[20];
    int year;
};

struct spers pers, pers1;
strcpy(pers.Name, "Иванов");
pers.year = 1960;

FILE *F;
if ((F = fopen("Test2.dat", "wb")) == NULL)
{
    ShowMessage("Файл не удастся создать");
    return;
}
fwrite(&pers, sizeof(spers), 1, F);
fclose(F);

if ((F = fopen("Test2.dat", "rb")) == NULL)
{
    ShowMessage("Файл не удастся открыть");
    return;
}
fread(&pers1, sizeof(spers), 1, F);
fclose(F);
```

13.9.2.4 Ввод/вывод, использующий дескрипторы

В C предусмотрен еще один механизм работы с файлами, основанный не на указателях на структуру типа **FILE**, а на дескрипторах. Файлы, открываемые подобным образом, не работают с буферами и с форматированными данными.

В начале работы любой программы автоматически открывается три потока со своими дескрипторами:

поток	дескриптор	
stdin	0	стандартный входной поток — обычно клавиатура
stdout	1	стандартный выходной поток — обычно экран
stderr	2	стандартный поток сообщений об ошибках

Но программа может и явным образом открывать любые новые файлы с дескрипторами.

Функции, работающие с дескрипторами файлов, описаны в файле `io.h`. Ряд используемых флагов и констант описан также в файлах `stdio.h`, `fcntl.h`, `sys\types.h` и `sys\stst.h`.

Файл открывается функцией `open`, которая возвращает дескриптор файла:

```
#include <fcntl.h>
#include<io.h>
int open(const char *path, int access, unsigned mode);
```

Параметр **path** указывает имя открываемого файла. Параметр **access** определяет режим доступа к файлу. Параметр **mode** является не обязательным и задает режим открытия файла.

Параметр **access** формируется операцией ИЛИ (`|`) из ряда флагов. Вот некоторые из них (полный список см. в главе 15 в разделе 15.5.1):

O_RDONLY	только для чтения
O_WRONLY	только для записи
O_RDWR	для чтения и записи
O_CREAT	создание нового файла
O_TRUNC	если файл существует, он урезается до 0
O_BINARY	двоичный файл
O_TEXT	текстовый файл

Параметр **mode** может принимать значения:

S_IWRITE	разрешение записи
S_IREAD	разрешение чтения
S_IREAD S_IWRITE	разрешение записи и чтения

Например, операторы

```
int handle;
if ((handle = open("Test.txt", O_CREAT | O_TEXT)) == -1)
{
    ShowMessage("Файл не удается создать");
    return;
}
```

пытаются создать новый текстовый файл, а в случае неудачи (функция `open` вернула `-1`) отображают сообщение об ошибке.

Имеется также функция `_creat`, осуществляющая примерно те же функции, что и `open`.

Закрывается файл функцией `close`:

```
int close(int handle);
```

Запись и чтение при работе с файлами, определяемыми дескрипторами **handle**, осуществляется функциями `write` и `read`:

```
#include <io.h>
int write(int handle, void *buf, unsigned len);
int read(int handle, void *buf, unsigned len);
```

В этих функциях **buf** — указатель на буфер, из которого записывается в файл или в который читается из файла **len** байтов.

Чтобы продемонстрировать работу с файлами, определенными своими дескрипторами, давайте воспроизведем пример, приведенный в разделе 13.9.2.3, в котором осуществлялась запись и чтение двух целых чисел **i** и **j**, действительного числа **a** и строки **s**:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <io.h>
#include <string.h>

int i = 1, j = 25, il, jl;
double a = 25e6, al;
char s[10], sl[10];
strcpy(s, "Иванов");

int handle;
// запись в файл
if ((handle = open("Test.txt", O_WRONLY | O_CREAT | O_BINARY)) == -1)
{
    ShowMessage("Файл не удается создать");
    return;
}
write(handle, &i, sizeof(int));           // запись i
write(handle, &j, sizeof(int));           // запись j
write(handle, &a, sizeof(double));        // запись a
write(handle, s, strlen(s)+1);            // запись строки s
close(handle);

// чтение из файла
if ((handle = open("Test.txt", O_RDONLY | O_BINARY)) == -1)
{
    ShowMessage("Файл не удается открыть");
    return;
}
read(handle, &il, sizeof(int));           // чтение i
read(handle, &jl, sizeof(int));           // чтение j
read(handle, &al, sizeof(double));        // чтение a
read(handle, sl, strlen(s)+1);            // чтение строки s
close(handle);
```

Если вы сравните этот код с тем, который был приведен в разделе 13.9.2.3, то увидите, что они практически идентичны и различаются только синтаксисом. Соответственно, и все приемы записи и чтения строк произвольной длины, рассмотренные в разделе 13.9.2.3, могут применяться и в данном случае.

Для работы с файлами, имеющими дескрипторы, могут использоваться функции **tell** и **lseek**, аналогичные рассмотренным в разделе 13.9.2.3 функциям **ftell** и **fseek**, производящими операции с указателями файлов. Имеются также функции **dup** и **dup2**, производящие операции непосредственно с дескрипторами, позволяющие создавать дубли дескрипторов или, например, перенаправлять стандартные потоки. Описания этих и иных функций вы найдете в главе 15 в разделе 15.5.3.

13.9.3 Файловый ввод/вывод с помощью потоков в стиле C++

13.9.3.1 Ввод и вывод потоков

В C++ определены три класса файлового ввода/вывода:

ifstream	входные файлы для чтения
ofstream	выходные файлы для записи
fstream	файлы для чтения и записи

При работе с файлами этих классов можно использовать ряд присущих им методов, но, пожалуй, основным достоинством использования этих классов является возможность применять очень удобные операции поместить в поток (<<) и взять из потока (>>).

Создаются объекты потоков, связанные с файлами, конструкторами соответствующих классов. Например, операторы

```
ofstream outfile("Test.dat");
if(!outfile)
{
    ShowMessage("Файл не удастся создать");
    return;
}
...
```

создают выходной поток **outfile**, связанный с файлом «Test.dat», создавая одновременно сам файл или, если он уже существует, урезая его длину до нуля. Если по каким-то причинам операция не может быть выполнена, значение **outfile** равно 0 и оператор **if** прерывает работу.

Аналогично может создаваться входной поток, связанный с файлом:

```
ifstream infile("Test.dat");
if(!infile)
{
    ShowMessage("Файл не удастся открыть");
    return;
}
...
```

К созданным таким образом потокам можно применять операции поместить в поток (<<) и взять из потока (>>), подробно рассмотренные в главе 12 в разделе 12.7.14. Преимущество этих операций, работающих с текстовыми файлами, по сравнению с рассмотренными в предыдущих разделах функциями является простота использования и автоматическое распознавание типов данных. Рассмотрим, например, следующий код:

```
int i = 1, j = 25, i1, j1;
double a = 25e6, a1;
char s[40], s1[40];
strcpy(s, "Иванов");

// создание файла как выходного потока
ofstream outfile("Test.dat");
if(!outfile)
{
    ShowMessage("Файл не удастся создать");
    return;
}
outfile << i << ' ' << j << ' ' << a << ' ' << s << endl;
```

```
// закрытие файла
outfile.close();

// открытие файла как входного потока
ifstream infile("Test.dat");
if(!infile)
{
    ShowMessage("Файл не удается открыть");
    return;
}
infile >> i1 >> j1 >> a1 >> s1;
// закрытие файла
infile.close();
```

В этом коде создается файл «Test.dat» и в него записываются в текстовом виде два целых числа **i** и **j**, действительное число **a** и строка **s**, содержащая одно слово, после чего манипулятором потока **endl** (см. раздел 12.7.14 главы 12) осуществляется перевод строки. Причем, запись всех этих данных осуществляется одним оператором, содержащим сцепленные операции поместить в поток. Если вы сравните это с аналогичными кодами, приведенными в предыдущих разделах, то убедитесь в компактности и простоте применения этой операции.

После того, как файл закроется, в нем будет записан текст «1 25 2.5e+07 Иванов». Дальнейшие операторы создают входной поток, связанный с этим файлом и одним оператором, содержащим сцепленные операции взять из потока читает все эти данные.

Особенности применения операций **<<** и **>>** детально рассмотрены в главе 12 в разделе 12.7.14. Отметим только, что возможности операции поместить в поток можно существенно расширить использованием манипуляторов потока, которые будут обсуждаться в следующем разделе 13.9.3.2. Помимо этой операции вывести данные в поток можно еще двумя способами: методом **put** и методом **write**.

Метод **put** выводит в поток один символ. Например, оператор

```
outfile.put('Я');
```

выведет в поток символ «Я». Функции **put** допускают сцепленный вызов. Например, оператор

```
outfile.put('Я').put('\n');
```

выведет в поток символ «Я» и символ перевода строки.

Метод **write** выводит в файл из символьного массива, на который указывает его первый параметр, число символов, указанных вторым параметром. Например, оператор

```
outfile.write(s, 5);
```

записывает в поток **outfile** 5 символов из массива **s**. Причем эти символы никак не обрабатываются, а просто выводятся в качестве сырых байтов данных. Среди этих символов, например, может встретиться в любом месте нулевой символ, но он не будет рассматриваться как признак конца строки.

Аналогичный метод **read** может затем прочитать эти символы в какой-то другой символьный массив и тоже без всякой обработки. Функция **gcount** сообщает о количестве символов, действительно прочитанных последней операцией ввода.

Теперь остановимся подробнее на вводе данных из файлового потока.

Операция взять из потока (**>>**) обладает особенностью, которую надо учитывать при вводе строк в массивы символов. Она читает не всю строку, а только одну лексему — последовательность символов до первого пробельного или разделительного символа. Иначе говоря, она читает не строку до символа перевода строки, а только одно слово. Это удобно, если надо производить анализ текста или искать в нем какое-то ключевое слово. Но это становится недостатком, если надо просто прочесть строку целиком.

В классе **ifstream** имеется еще два метода чтения из потока: **get** и **getline**. Метод **get** имеет три модификации: **get()**, **get(char)** и **get(char *, int n, char delim)**.

Функция **get** без аргументов вводит одиночный символ из указанного потока (даже, если это символ разделитель) и возвращает этот символ в качестве значения вызова функции. Этот вариант функции **get** возвращает **EOF**, когда в потоке встречается признак конца файла.

Следующий код использует функцию **get** без аргумента, чтобы построчно читать и обрабатывать весь текст файла:

```
char s[80], c;
ifstream infile("Test.dat");
if(!infile)
{
    ShowMessage("Файл не удастся открыть");
    return;
}
int i = 0;
while((c = infile.get()) != EOF)
{
    if(c == '\n')
    {
        // занесение нулевого символа в конец строки
        s[i] = 0;
        // обработка строки
        ...
        i = 0;
    }
    // формирование строки
    else s[i++] = c;
}
// закрытие файла
infile.close();
```

Здесь символы файла поочередно читаются в символьную переменную **c**. Если прочитанный символ не является символом перевода строки «**/n**», то символ добавляется в строку **s**. Если же символ равен «**/n**», то в конец строки заносится нулевой символ строка подвергается какой-то обработке, после чего начинает формироваться следующая строка. Отметим, что этот код имеет один недостаток: если символу конца файла не предшествует символ перевода строки, то последняя строка оказывается без завершающего нулевого символа и остается необработанной. Нетрудно придумать дополнение кода, которое ликвидировало бы этот недостаток.

Функцию **get()** удобно использовать для поиска в файле какого-то ключевого символа. Например, цикл поиска в файле символа «**\$**» можно организовать следующим образом:

```
while((c = infile.get()) != EOF)
    if(c == '$') break;
if(c == '$') ...
```

Другой вариант функции-элемента **get** с символьным аргументом вводит очередной символ из входного потока (даже, если это символ разделитель) и сохраняет его в символьном аргументе. Этот вариант функции **get** возвращает ложь, когда встречается признак конца файла; в остальных случаях этот вариант функции **get** возвращает ссылку на тот объект потока, для которого вызывалась функция-элемент **get**.

При использовании этого варианта функции **get** приведенные ранее примеры можно оставить практически без изменений, переписав только заголовки структур **while**:

```
while(infile.get(c))
```

Третий вариант функции-элемента **get** принимает три параметра: символьный массив **s**, максимальное число символов **n** и ограничитель **delim** (по умолчанию символ перевода строки `'\n'`). Этот вариант читает символы из входного потока до тех пор, пока не достигается число символов, на 1 меньше указанного максимального числа **n**, или пока не считывается ограничитель. Затем для завершения введенной строки в символьный массив, используемый в качестве буфера программы, помещается нулевой символ. Ограничитель в символьный массив не помещается, а остается во входном потоке (он будет следующим считываемым символом). Таким образом, результатом второго подряд использования функции **get** явится пустая строка, если только ограничитель не удалить из входного потока.

Приведенный ранее пример чтения всего файла по строкам в данном случае реализуется проще:

```
char s[80];
ifstream infile("Test.dat");
if(!infile)
{
    ShowMessage("Файл не удастся открыть");
    return;
}
while(!infile.eof())
{
    infile.get(s,80);
    infile.get();
    // обработка строки
    ...
}
// закрытие файла
infile.close();
```

В данном случае третий аргумент в вызове **get** не указан. Значит подразумевается по умолчанию ограничитель `«\n»` и каждый вызов **get** читает одну строку (подразумевается, что ее длина не более 80 символов). Обратите внимание на то, что после оператора

```
infile.get(s,80);
```

добавлен оператор

```
infile.get();
```

Этот оператор удаляет из потока ограничитель. Если этого не сделать, программа заикнется.

Функция **get** с тремя параметрами не всегда удобна, поскольку оставляет ограничитель в потоке, и для повторного вызова функции его приходится убирать отдельным оператором. Часто более удобна другая функция — **getline**. Эта функция действует подобно третьему варианту функции **get** и помещает нулевой символ после строки в символьном массиве. Но в отличие от **get** функция **getline** удаляет символ ограничитель из потока (т.е. читает этот символ и отбрасывает его); этот символ не сохраняется в символьном массиве.

С помощью **getline** рассмотренный выше цикл чтения файла по строкам может быть записан следующим образом:

```
while(!infile.eof())
{
    infile.getline(s,80);
    // обработка строки
    ...
}
```

13.9.3.2 Манипуляторы потоков

В разделе 13.9.3.1 и в главе 12 в разделе 12.7.14 рассматривался один из манипуляторов потоков — манипулятор **endl**, переводящий поток на новую строку. Имеется еще много манипуляторов потока, позволяющих форматировать вывод в файл операцией вывода в поток <<.

Чтобы посмотреть возможности манипуляторов, вы можете построить приложение, аналогичное рассмотренным в предыдущих разделах и содержащее окно **Memo1** и кнопку, обработчик события **OnClick** которой имеет следующий вид:

```
#include <fstream.h>
#include <iomanip.h>

char s[40];

// создание файла как выходного потока
ofstream outfile("Test.dat");
if(!outfile)
{
    ShowMessage("Файл не удается создать");
    return;
}
// операторы, использующие операция вывести в поток
...
// закрытие файла
outfile.close();

ifstream infile("Test.dat");
if(!infile)
{
    ShowMessage("Файл не удается открыть");
    return;
}
Memo1->Clear();
while(!infile.eof())
{
    infile.getline(s,80);
    Memo1->Lines->Add(s);
}
// закрытие файла
infile.close();
```

Манипуляторы **dec**, **oct**, **hex** и **setbase** определяют систему счисления, в которой выводятся целые числа — соответственно десятичную, восьмеричную, шестнадцатеричную и с заданным основанием. По умолчанию целые числа выводятся как десятичные. Послав в поток один из перечисленных модификаторов вы можете перейти к другой системе счисления и она будет действовать до тех пор, пока вы не примените новый модификатор. Модификатор **setbase** относится к параметризованным модификаторам потоков. В качестве параметра в него передается основание системы счисления. Для применения этого и других параметризованных модификаторов надо включить в проект заголовочный файл **<iomanip.h>**. Приведем пример использования рассмотренных модификаторов. Операторы

```
int i = 31;
outfile << i << ' ' << hex << i << ' ' << oct << i << ' '
    << setbase(10) << i << endl;
```

приведут к записи текста «31 1f 37 31». Сначала число 31 отображается в десятичном виде, потом в восьмеричном, затем в шестнадцатеричном и в заключение опять в десятичном.

Можно управлять точностью выводимых чисел с плавающей запятой, т.е. числом разрядов справа от десятичной точки, используя манипулятор потока **setprecision** или метод **precision**. Вызов любой из этих установок точности действует для всех последующих операций вывода до тех пор, пока не будет произведена следующая установка точности.

Чтобы посмотреть возможности способов управления точностью, вы можете записать оператор:

```
for(int i = 0; i < 10; i++)
    outfile << setprecision(i) << sqrt(3.0) << endl;
```

Этот оператор изменяет в цикле параметр манипулятора **setprecision** от 0 до 9 и тем самым изменяет точность вывода в файл значения корня квадратного из 3. Значение параметра 0 приводит к установке точности по умолчанию, которая равна 6. Результат работы приведенного оператора следующий:

```
1.73205
2
1.7
1.73
1.732
1.7321
1.73205
1.732051
1.7320508
1.73205081
```

Аналогичный результат даст следующий код, использующий метод **precision**:

```
for(int i = 0; i < 10; i++)
{
    outfile.precision(i);
    outfile << sqrt(3.0) << endl;
}
```

Если в функции **precision** не задавать параметров, например,

```
int i = outfile.precision();
```

то она вернет текущую установку точности.

Манипулятор потока **setw** и метод **width** устанавливают ширину поля (т.е. число символьных позиций, в которые значение будет выведено, или число символов, которые будут введены) и возвращает предыдущую ширину поля. Если обрабатываемые значения имеют меньше символов, чем заданная ширина поля, то для заполнения лишних позиций используются заполняющие символы. По умолчанию заполняющими символами являются пробелы и вставляются они перед значащими символами, т.е. происходит выравнивание вправо. Если число символов в обрабатываемом значении больше, чем заданная ширина поля, то лишние символы не отсекаются и число будет напечатано полностью. Установка ширины поля влияет только на следующую операцию поместить в поток; затем ширина поля устанавливается неявным образом на 0, т.е. поле для представления выходных значений будут просто такой ширины, которая необходима. Функция **width**, не имеющая аргументов, возвращает текущую установку ширины поля.

Заполняющие символы могут устанавливаться манипулятором **setfill(char)** или методом **fill**.

Например, следующие операторы демонстрируют влияние ширины поля на результат вывода числа 25:

```
int j = 25;
for(int i = 0; i < 5; i++)
    outfile << setw(i) << j << endl;
```

Результат работы этих операторов следующий:

```
25
25
25
 25
 25
```

Из этого результата видно, что пока ширина поля меньше числа символов в выводимом числе, она ни на что не влияет, а при большой ширине поля происходит выравнивание числа вправо.

Такой же результат дает и следующий цикл, использующие метод **width**:

```
for(int i = 0; i < 5; i++)
{
    outfile.width(i);
    outfile << j << endl;
}
```

Если вывести в поток модификатор **setfill**, то заполняющие символы изменятся. Например, оператор

```
for(int i = 0; i < 5; i++)
    outfile << setfill('*') << setw(i) << j << endl;
```

приведет к результату:

```
25
25
25
*25
**25
```

Мы рассмотрели многие (но еще не все) манипуляторы потоков. Пользователи могут создавать собственные манипуляторы потоков. В качестве примера того, как это делается, ниже приводится код функции, создающей манипулятор, названный **tab**, который выводит в поток символ табуляции «\t».

```
//Создание манипулятора tab
ostream& tab(ostream& output)
{
    return output << '\t';
}
```

Если вы ввели в приложение такую функцию, то в дальнейшем можете использовать этот манипулятор. Например, оператор

```
outfile << 'A' << tab << 'B' << tab << 'C' << endl;
```

выведет символы «А», «В» и «С», разделенные символами табуляции:

```
A   B   C
```

13.9.3.3 Флаги состояния формата

В классе **ios** — базовом классе всех потоков ввода/вывода определены следующие флаги формата.

ios::skipws	пропуск символов разделителей во входном потоке
ios::left	выравнивание по левой границе поля
ios::right	выравнивание по правой границе поля
ios::internal	выравнивание знака или основания системы счисления по левой, а числа — по правой границам поля

ios::dec	десятичная система счисления, устанавливается манипулятором dec
ios::oct	восьмеричная система счисления, устанавливается манипулятором oct
ios::hex	шестнадцатеричная система счисления, устанавливается манипулятором hex
ios::showbase	вывод основания системы счисления
ios::showpoint	обязательная печать десятичной точки и нулевых младших разрядов
ios::uppercase	вывод в верхнем регистре символов X и E в шестнадцатеричном и экспоненциальном форматах
ios::showpos	вывод символа «+» перед положительным числом
ios::scientific	экспоненциальное представление действительных чисел
ios::fixed	формат действительных чисел с фиксированной точкой

Флаги состояния формата управляются методами **flags**, **setf** и **unsetf**, или манипуляторами потоков **setw**, **setiosflags** и **resetiosflags**.

Метод **flags** используется для задания сразу всех флагов. При этом те флаги, которые должны быть установлены, объединяются операцией поразрядного ИЛИ (|) в одно значение типа **long**, передаваемое методу как параметр. Метод **flags** возвращает значение типа **long**, содержащее предыдущие значения опций. Это значение часто сохраняется с тем, чтобы можно было впоследствии вызвать функцию **flags** с этим сохраненным значением и восстановить предыдущие значения опций.

Метод **setf** и параметризованный манипулятор потока **setiosflags** имеют единственный аргумент, который устанавливает один или более флагов, соединенных операцией |, и может использовать текущие установки флагов для создания нового состояния формата. Например, манипулятор

```
setiosflags(ios::showpos | ios::showpoint)
```

устанавливает флаги **ios::showpos** и **ios::showpoint**.

Манипулятор потока **resetiosflags** и метод **unsetf** наоборот, сбрасывают флаги, которые указаны их параметром. Чтобы использовать перечисленные параметризованные манипуляторы потока, надо в приложение включить директиву **#include <iomanip.h>**.

Приведем примеры использования перечисленных флагов состояния формата.

Оператор, использующий флаг **showpoint**:

```
outfile << 1. << " " << 1.1 << " " <<
    setiosflags(ios::showpoint) << 1. << " " << 1.1 << endl;
```

дает результат:

```
1 1.1 1.00000 1.10000
```

Оператор, использующий флаги **right**, **left** и **internal**:

```
outfile << setw(6) << -1.1 << endl
    << setw(6) << resetiosflags(ios::right)
    << setiosflags(ios::left) << -1.1 << endl
    << setw(6) << resetiosflags(ios::left)
    << setiosflags(ios::internal) << -1.1 << endl;
```

дает результат:

```
-1.1
-1.1
- 1.1
```

Обратите внимание на то, что надо сбрасывать модификатором **resetiosflags** ранее установленный флаг, чтобы при каждом выводе только один из флагов **right**, **left** и **internal** был установлен.

Оператор, использующий флаг **showbase**:

```
outfile << 63 << oct << " " << 63 << hex << " " << 63
      << setiosflags(ios::showbase) << dec << endl
      << 63 << oct << " " << 63 << hex << " " << 63 << endl;
```

дает результат:

```
63 77 3f
63 077 0x3f
```

Обратите внимание, что флаги системы счисления устанавливаются не модификатором **setiosflags**, а модификаторами **dec**, **oct**, **hex**.

Оператор, использующий флаги **scientific** и **fixed**:

```
outfile << "По умолчанию:" << endl
      << 0.0123 << ' ' << 1.23e6 << endl << endl
      << "Флаг scientific:" << setiosflags(ios::scientific)
      << endl
      << 0.0123 << ' ' << 1.23e6 << endl << endl
      << "Флаг fixed:" << resetiosflags(ios::scientific)
      << setiosflags(ios::fixed) << endl
      << 0.0123 << ' ' << 1.23e6 << endl;
```

дает результат:

```
По умолчанию:
0.0123 1.23e+06

Флаг scientific:
1.230000e-02 1.230000e+06

Флаг fixed:
0.012300 1230000.000000
```

Обратите внимание, что по умолчанию значения чисел с плавающей запятой сами выбирают формат представления и он, пожалуй, наиболее привлекателен.

Оператор, использующий флаги **showpos** и **showpoint**:

```
outfile << setprecision(4) << setw(3) << 60. << endl
      << setiosflags(ios::showpos | ios::showpoint) << 60.
      << endl;
```

дает результат:

```
60
+60.00
```

В этом примере один манипулятор **setiosflags** устанавливает сразу два флага.

13.10 Массивы

13.10.1 Одномерные массивы

Массив представляет собой структуру данных, позволяющую хранить под одним именем совокупность данных любого, но только одного какого-то типа. Массив характеризуется своим именем, типом хранимых элементов, размером (числом

хранимых элементов), нумерацией элементов и размерностью. В данном разделе мы ограничимся одномерными массивами, т.е. массивами с размерностью 1.

Объявление переменной как одномерного массива имеет вид:

```
тип переменная [константное_выражение]
```

Например, оператор

```
int A[10];
```

объявляет массив с именем **A**, содержащий 10 целых чисел. Доступ к элементам этого массива осуществляется выражением **A[i]**, где **i** — индекс, являющийся в данном примере, как видно из объявления, целым числом в диапазоне 0 - 9. Например, **A[0]** — значение первого элемента, **A[1]** — второго, **A[9]** — последнего. Обратите внимание, что индекс последнего элемента на 1 меньше размера массива. Это связано с тем, что индексы начинаются с 0.

Приведем примеры использования этого массива. Код

```
A[0] = 1;
A[1] = 1;
for(int i = 2; i < 10; i++) A[i] = A[i-2]+A[i-1];
```

заполняет массив так называемыми числами Фибоначчи, первые 2 из которых равны 1, а каждое последующее равно сумме двух предыдущих.

Элементы массива могут иметь любой тип. Например, предложение

```
char S[10];
```

объявляет массив символов. Массив символов — это фактически строка (см. раздел 13.4.1) и с ним можно во многом обращаться как со строкой, хотя можно обращаться и как с массивом. При использовании массива символов как строки надо только иметь в виду, что это строка фиксированной допустимой длины. И число символов, помещаемых в строку, не должно превышает объявленного размера массива **n-1**, поскольку строка кончается нулевым символом.

Объявление переменной массива можно совмещать с заданием элементам массива начальных значений. Эти значения перечисляются в списке инициализации после знака равенства, разделяются запятыми и заключаются в фигурные скобки. Например:

```
int A[10] = {1,2,3,4,5,6,7,8,9,10};
char S[10] = {"abcdefghi\0"};
```

Если начальных значений меньше, чем элементов в массиве, оставшиеся элементы автоматически получают нулевые начальные значения. Например, оператор

```
int A[10] = {1,2,3};
```

задает значения первым трем элементам, а остальные будут равны 0. Оператор

```
int A[10] = {0};
```

присваивает нулевые значения всем элементам массива.

Предупреждение

Если массив при его объявлении не инициализирован, то его элементы имеют случайные значения. Элементы такого массива нельзя использовать в выражениях, пока им не будут присвоены какие-нибудь значения.

В массивах символов задание нулей элементам, не указанным в списке инициализации, равносильно заданию нулевых символов, означающих конец строки. Поэтому приведенное выше объявление переменной **S** с ее инициализацией избы-

точно. Нулевой символ в конце можно не указывать. Например, нормально будут восприняты такие объявления:

```
char S[10] = {"abcdefghi"};
char S1[10] = {"abc"};
```

Последнее объявление выделяет место под массив из 10 элементов, но инициализирует его строкой из трех элементов.

В объявлении со списком инициализации размер массива можно не указывать. Тогда количество элементов массива будет равно количеству элементов в списке начальных значений. Например, объявление

```
int A[ ] = {1, 2, 3, 4, 5};
```

создает массив из пяти элементов. Объявление

```
char S1[ ] = {"abc"};
```

создает массив из четырех элементов — три значащих символа плюс нулевой символ.

В объявлении массива в качестве размера лучше всегда использовать именованные константы. Например, ниже приведено объявление массива и оператор, подсчитывающий сумму его элементов:

```
int A[10];
// операторы заполнения массива
...
// подсчет суммы
int Sum = A[0];
for(int i = 1; i < 10; i++) Sum += A[i];
```

Если в дальнейшем вы решите, что вам требуется массив **A** не из 10 элементов, а, например, из 100, вы должны будете изменить размер массива и в объявлении **A**, и во всех операторах, работающих с этим массивом (в данном случае в операторе **for**). А ведь таких операторов в разных частях программы может быть очень много. О такой программе говорят, что она плохо масштабируется.

Грамотнее реализовать этот пример следующим образом:

```
const Amax = 10;
int A[Amax];
// операторы заполнения массива
...
// подсчет суммы
int Sum = A[0];
for(int i = 1; i < Amax; i++) Sum += A[i];
```

В этом случае вы вводите именованную константу **Amax** и используете ее во всех операторах, в которых вам требуется размер массива. Тогда при необходимости изменить размер массива вам достаточно изменить его только в одном операторе, объявляющем **Amax**. Программа сразу становится масштабируемой. А объявление **Amax** как константы гарантирует, что объявленное значение не будет случайно изменено где-то в программе.

Аналогичный результат можно получить, если заменить объявление константы директивой компилятора **#define** (см. главу 12 раздел 12.2.2).

```
#define Amax 10
```

Хороший стиль программирования

Все размеры массивов в программе следует определять именованными константами или макросами. Это делает программу более понятной и существенно облегчает ее отладку и сопровождение.

В ряде случаев требуются константные массивы, данные из которых программа может только читать. Такие массивы обязательно должны инициироваться в момент объявления. Например:

```
const AnsiString Day[] = {"понедельник", "вторник", "среда", "четверг",
                          "пятница", "суббота", "воскресенье"};
```

13.10.2 Многомерные массивы

Можно объявлять и многомерные массивы, т.е. массивы, элементами которых являются массивы. Например, двумерный массив можно объявить таким образом:

```
int A2[10][3];
```

Этот оператор описывает двумерный массив, который можно представить себе как таблицу, состоящую из 10 строк и 3 столбцов.

Доступ к значениям элементов многомерного массива обеспечивается через индексы, каждый из которых заключается в квадратные скобки. Например, **A2[3][2]** — значение элемента, лежащего на пересечении четвертой строки и третьего столбца (помните, что индексы начинаются с 0).

Если многомерный массив инициализируется при его объявлении, список значений по каждой размерности заключается в фигурные скобки. Приведенный ниже оператор объявляет трехмерный массив **A3** размерностью 4 на 3 на 2.

```
int A3[4][3][2] = {{{0,1},{2,3},{4,5}},
                    {{6,7},{8,9},{10,11}},
                    {{12,13},{14,15},{16,17}},
                    {{18,19},{20,21},{22,23}}};
```

Этот оператор создает массив **A3**, четыре строки которого являются матрицами вида

0	1	6	7	12	13	18	19
2	3	8	9	14	15	20	21
4	5	10	11	16	17	22	23

Например, элемент **A3[0][1][0]** равен 2, элемент **A3[3][0][1]** равен 19 и т.д.

Если в списке инициализации в какой-то из размерностей не хватает данных, то все дальнейшие не перечисленные элементы считаются равными нулям.

13.10.3 Операции с массивами, передача массивов как параметров

Имя массива является константным указателем на первый элемент массива. Взаимосвязь массивов и указателей подробно рассмотрена в разделе 13.7. Поскольку имя массива — константный указатель, оно не может модифицироваться и к нему не применимы все операции присваивания.

К имени массива можно применять операцию **sizeof**, которая в этом случае возвращает значение, равное общему объему памяти, отведенному под все элементы массива. Таким образом, число элементов массива **A** можно определить выражением

```
sizeof(A) / sizeof(A[0])
```

поскольку под каждый элемент массива отведен одинаковый объем памяти.

Подобное вычисление размера массива выполняет макрос **ARRAYSIZE**. Приведенное выше выражение эквивалентно выражению

```
ARRAYSIZE(A)
```

При передаче массива в функцию в качестве параметра заголовок функции содержит тип и имя массива с последующими пустыми квадратными скобками. Например, если функция **F** должна принимать массив как параметр, ее прототип может иметь вид:

```
void F(int Ar[]);
```

Обращение к такой функции может быть записано так:

```
const Amax = 10;
int A[Amax];
...
F(A);
```

Как видно, в вызове функции указывается просто имя массива. Внутри функции к элементам этого массива можно обращаться обычным образом, например, **Ar[2]**. C++ передает имя массива в функцию по ссылке (см. главу 12 раздел 12.5.2). Это значит, что если функция изменяет значения элементов массива, то изменяются элементы исходного массива, который передавался в функцию.

В большинстве случаев только имени массива мало, чтобы провести в функции обработку его элементов. Внутри функции требуется знать размер массива, чтобы можно было организовать его циклическую обработку. Поэтому обычно в функцию передается не только массив, но и его размер. При этом заголовок функции может иметь вид:

```
void F(int Ar[], int N);
```

а вызов функции:

```
F(A, Amax);
```

Чаше библиотечные функции требуют в качестве второго параметра не размер массива, а значение его последнего индекса, которое на единицу меньше размера. В этом случае вызов функции может иметь вид:

```
F(A, Amax - 1);
```

В частности, такого вызова требуют все функции Object Pascal, использующие так называемый открытый массив. Поскольку подобные вызовы функции встречаются довольно часто, в файле **sysdefs.h** определен макрос **EXISTINGARRAY**, который позволяет оформить передачу массива более компактно. При использовании этого макроса приведенный выше вызов можно оформить так:

```
F(EXISTINGARRAY(A));
```

При разворачивании макрос **EXISTINGARRAY** передаст в функцию имя массива как первый параметр и значение последнего индекса как второй параметр. При этом макрос использует приведенное ранее выражение для подсчета числа элементов массива через операцию **sizeof**.

Некоторые функции Object Pascal, используемые и в C++Builder, могут воспринимать в качестве параметров так называемые открытые массивы констант, в которых могут содержаться элементы разных типов. В файле **sysdefs.h** описан макрос **OPENARRAY**, позволяющий обращаться к таким функциям. Без дополнительных разъяснений приведем форму записи такого макроса:

```
OPENARRAY(TVarRec, (элемент_1, элемент_2, ...))
```

Число передаваемых элементов может достигать 19.

Передача массива по ссылке не гарантирует защиты от несанкционированного изменения программой значений элементов массива. Если необходимо защитить массив от подобных изменений, его надо передать в функцию как константный:

```
void F(const int Ar[], int N);
```

Пусть, например, вы хотите написать функцию, подсчитывающую сумму элементов массива целых. Тогда вы можете оформить ее следующим образом:

```
int Sum(const int A[], int N)
{
    // N - размер массива
    int S = A[0];
    for(int i = 1; i < N; i++) S += A[i];
    return S;
}
```

Ниже приведен пример тестирования этой функции.

```
#define Bmax 10
int B[10] = {1,2,3,4,5,6,7,8,9,10};
ShowMessage("Сумма равна " + IntToStr(Sum(B,Bmax)));
```

При вызове функции не обязательно передавать весь массив. Можно передать только какую-то его часть. Например, вы можете передать в функцию параметр размера массива, меньший истинного. Если вы в приведенном примере в качестве второго параметра передадите в функцию не **Bmax**, а **Bmax - 2**, то функция будет обрабатывать только восемь первых элементов с индексами от 0 до 7. Можно и в качестве начала массива передать в функцию указатель на какой-то элемент массива. Например, если вы обратитесь к функции так:

```
Sum(B + 2, Bmax - 2)
```

то вы передадите в нее указатель не на первый, а на третий элемент. Поэтому, когда функция будет обращаться к элементам массива от 0 до 7, в действительности она будет работать с элементами, индексы которых от 2 до 9. Т.е. сумма будет посчитана по элементам, начиная с третьего.

Если в функцию передается многомерный массив, то в заголовке только квадратные скобки первой размерности остаются пустыми, а в скобках следующих размерностей должны указываться константами их размеры. Например, если функция **F2** должна принимать двумерный массив размером 3 на 3, то ее заголовок может иметь вид:

```
void F(const int Ar[][3]);
```

Вызов этой функции производится обычной передачей в нее имени массива. Например, **F(A)**.

13.11 Структуры

13.11.1 Структуры в стиле C

Структуры — это составные типы данных, построенные с использованием других типов. Они представляет собой объединенный общим именем набор данных различных типов. Именно тем, что в них могут храниться данные разных типов, они и отличаются от массивов, хранящих данные одного типа.

Отдельные данные структуры называются элементами или полями. Все это напоминает запись в базе данных, только хранящуюся в оперативной памяти компьютера.

Простейший вариант объявления структуры может выглядеть следующим образом:

```
struct TPers {
    AnsiString Fam,Nam,Par;
    unsigned   Year;
    bool       Sex;
    AnsiString Dep;
};
```


Ключевое слово **struct** начинает определение структуры. Идентификатор **TPers** — тег (обозначение, имя-этикетка) структуры. Тег структуры используется при объявлении переменных структур данного типа. В этом примере имя нового типа — **TPers**. Имена, объявленные в фигурных скобках описания структуры — это элементы структуры. Элементы одной и той же структуры должны иметь уникальные имена, но две разные структуры могут содержать не конфликтующие элементы с одинаковыми именами. Каждое определение структуры должно заканчиваться точкой с запятой.

Определение **TPers** содержит шесть элементов. Предполагается, что такая структура может хранить данные о сотруднике некоего учреждения. Типы данных разные: элементы **Fam**, **Nam**, **Par** и **Dep** — строки, хранящие соответственно фамилию, имя, отчество сотрудника и название отдела, в котором он работает. Элемент **Year** целого типа хранит год рождения, элемент **Sex** булева типа хранит сведения о поле. Элементы структуры могут быть любого типа, но структура не может содержать экземпляры самой себя. Например, элемент типа **TPers** не может быть объявлен в определении структуры **TPers**. Однако, может быть включен указатель на другую структуру типа **TPers**. Структура, содержащая элемент, который является указателем на такой же структурный тип, называется структурой с самоадресацией. Такие структуры очень полезны для формирования различных списков (см. раздел 13.11.2).

Само по себе объявление структуры не резервирует никакого пространства в памяти; оно только создает новый тип данных, который может использоваться для объявления переменных. Переменные структуры объявляются так же, как переменные других типов. Объявление

```
TPers Pers, PersArray[10], *Ppers;
```

объявляет переменную **Pers** типа **TPers**, массив **PersArray** — с 10 элементами типа **TPers** и указатель **Ppers** на объект типа **TPers**.

Переменные структуры могут объявляться и непосредственно в объявлении самой структуры после закрывающейся фигурной скобки. В этом случае указание тега не обязательно:

```
struct {
    AnsiString Fam,Nam,Par;
    unsigned   Year;
    bool       Sex;
    AnsiString Dep;
}Pers, PersArray[10], *Ppers;
```

Для доступа к элементам структуры используются операции доступа к элементам: операция точка (.) и операция стрелка (->). Операция точка обращается к элементу структуры по имени объекта или по ссылке на объект. Например:

```
Pers.Fam = "Иванов";
Pers.Nam = "Иван";
Pers.Par = "Иванович";
Pers.Year = 1960;
Pers.Sex = true;
Pers.Dep = "Бухгалтерия";
```

Операция стрелка обеспечивает доступ к элементу структуры через указатель на объект. Допустим, что выполнен оператор

```
Ppers = &Pers;
```

который присвоил указателю **Ppers** адрес объекта **Pers**. Тогда указанные выше присваивания элементам структуры можно выполнить так:

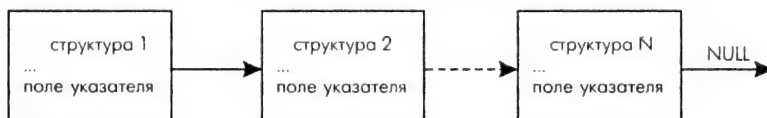
```
Ppers->Fam = "Иванов";
Ppers->Nam = "Иван";
Ppers->Par = "Иванович";
```

```
Ppers->Year = 1960;
Ppers->Sex = true;
Ppers->Dep = "Бухгалтерия";
```

13.11.2 Самоадресуемые структуры

Теперь рассмотрим еще один вид структур — самоадресуемые структуры. Нередко в памяти надо динамически размещать (см. главу 12 раздел 12.9) последовательность структур, как бы формируя некий фрагмент базы данных, предназначенный для оперативного анализа и обработки. Поскольку динамическое размещение проводится в непредсказуемых местах памяти, то такие структуры надо снабдить элементами, содержащими указатели на следующую аналогичную структуру. Такие структуры со ссылками на аналогичные структуры и называются самоадресуемыми. Ниже приведена схема связи таких структур в последовательность. Полю указателя в последней структуре обычно присваивается значение **NULL**, что является признаком последней структуры при организации поиска в списке.

Рис. 13.1
Список структур



Если мы хотим структуру, рассмотренную в разделе 13.11.1, сделать самоадресуемой, следует изменить ее объявление следующим образом:

```
struct TPers {
    AnsiString Fam, Nam, Par;
    unsigned   Year;
    bool       Sex;
    AnsiString Dep;
    TPers * pr;
};
```

Приведем пример формирования в памяти списка таких структур. Для этого надо определить три переменные, являющиеся указателями на структуры:

```
TPers *P0 = NULL, *Pnew, *Pold;
```

Первая из этих переменных будет всегда указывать на первую структуру в списке. Две остальные переменные — вспомогательные. Если в некоторый момент возникла необходимость динамически разместить в памяти очередную структуру и вставить ее в конец списка, это можно сделать следующим кодом:

```
// Выделение памяти под новую структуру
Pnew = new TPers;

// Заполнение элементов структуры
Pnew->Fam = "O>BRT>";
Pnew->Nam = "O>BR";
Pnew->Par = "O>BRT>P)";
Pnew->Year = 1960;
Pnew->Sex = true;
Pnew->Dep = "Бухгалтерия";
Pnew->pr = NULL;

if(P0 == NULL) P0 = Pnew; // P0 - указатель на первую структуру
else Pold->pr = Pnew;     // указатель на очередную структур
Pold = Pnew;
```

Если список еще не начат (**P0** = **NULL**), то указателю **P0** присваивается ссылка на вновь размещенную структуру (**Pnew**). В противном случае ссылка на новую структуру присваивается полю **pr** предыдущей структуры в списке (**Pold**). Таким образом новая структура включается в общий список. Полю **pr** этой структуры присваивается значение **NULL**. Это является признаком того, что данная структура является последней в списке.

Сформировав список в памяти далее легко его просматривать, проходя в цикле по указателям. Например:

```
Pnew = P0;
while(Pnew != NULL)
{
    ShowMessage(Pnew->Fam + " " + Pnew->Nam + " " + Pnew->Par);
    Pnew = Pnew->pr;           // переход к новой структуре
}
```

Легко также делать в списке перестановки структур, их удаление и т.п. Для всех этих операций не надо ничего перемещать в памяти. Достаточно только изменять соответствующие ссылки в полях **pr**.

Раньше подобные списки широко использовались для создания в памяти стеков, очередей и других упорядоченных списков. Однако, в C++Builder введены специальные типы данных **TList** и **TStringList**, которые ведут подобные списки и имеют множество удобных методов для управления ими. Изучите эти типы, рассмотренные в главе 16.

13.11.3 Структуры в стиле C++

Все, что рассмотрено в предыдущих разделах, относится как к языку C, так и к C++. Но в C++ понятие структуры существенно расширено и приближено к понятию класса (см. раздел 13.13).

В частности, в структурах кроме рассмотренных ранее данных-элементов разрешается описывать функции-элементы. Рассмотрим это на примере использованной в предыдущих разделах структуры **TPers**. Давайте введем в эту структуру функцию-элемент **Show**, отображающую информацию, хранящуюся в структуре:

```
struct TPers {
    AnsiString Fam, Nam, Par;
    unsigned   Year;
    bool       Sex;
    AnsiString Dep;
    TPers * pr;
    void Show()
    {
        ShowMessage("Сотрудник отдела \""+Dep+"\" "+Fam+" "+Nam+" "+
            Par+", "+IntToStr(Year)+" г.р., пол "+
            (Sex ? "мужской" : "женский"));
    }
};
```

Функция **Show** отображает информацию вида: «Сотрудник отдела «Бухгалтерия» Иванов Иван Иванович, 1960 г.р., пол мужской».

Обращение к этой функции-элементу производится через переменную структуры операцией точка или через указатель на переменную операцией стрелка. Например:

```
Pers.Show();
Pnew->Show();
```

С использованием введенной функции **Show** приведенный в разделе 13.11.2 пример просмотра списка можно упростить:

```
Pnew = P0;
while(Pnew != NULL)
{
    Pnew->Show();
    Pnew = Pnew->pr;
}
```

В C++ можно вводить спецификаторы доступа к данным-элементам и функциям-элементам так же, как это делается в классе. Разрешаются спецификаторы **public** (открытый) и **private** (закрытый). Закрытые элементы структуры могут быть доступны только для функций-элементов этой структуры. Ни через объект, ни через указатель на объект доступ к ним невозможен. Закрытыми объявляются какие-то вспомогательные данные-элементы, не представляющие интереса для пользователя, а также вспомогательные функции (утилиты), требующиеся для работы основных функций-элементов структуры.

Открытые элементы структуры могут быть доступны для любых функций в программе. Основная задача открытых элементов состоит в том, чтобы дать клиентам структуры представление о возможностях, которые она имеет. Это открытый интерфейс структуры.

По умолчанию доступ к элементам структуры **public** — открытый. Если вам надо спрятать от пользователя какие-то элементы, укажите спецификатор **private**, завершающийся двоеточием, и помещайте после него объявления закрытых элементов. Все, что помещено после спецификатора **private** до конца структуры или до спецификатора **public**, будет скрыто от пользователя. Например, в следующем объявлении структуры

```
struct MyStr {
    int x, y;
    int Get();
private:
    int a, b;
    void F();
};
```

данные **x** и **y** и функция **Get** — открытые и могут использоваться при работе со структурой, а данные **a** и **b** и функция **F** — закрытые и ими может пользоваться только функция **Get**.

Есть еще ряд особенностей, сближающих в C++ структуры и классы. Они будут рассмотрены в разделе 13.13, посвященном классам.

13.11.4 Битовые поля

Язык Си++ предоставляет возможность задавать количество битов, в которых хранятся элементы типов **unsigned** или **int** структуры (а также класса и объединения — см. разделы 13.13 и 13.12). Такие элементы называются битовыми полями. Битовые поля позволяют рационально использовать память с помощью хранения данных в минимально требуемом количестве битов.

В структуре **TPers**, использовавшейся в предыдущих разделах, можно, например, сократить затраты на хранение года рождения и пола сотрудника. Если ориентироваться на даты до 2047 года, то для хранения года рождения достаточно 11 битов. Если вы рассчитываете, что ваша программа просуществует дольше, то можете даже выделить под год 12 битов — этого хватит на ближайшие две тысячи лет. А под хранение сведений о поле вполне достаточно 1 бита. Таким образом, под эти два элемента вам достаточно 2 байтов, а в описанной ранее версии структуры под эти элементы отводилось 5 байтов: 4 под год плюс один под пол. Выигрыш 3 байта. Конечно, немного, но если при выполнении вашей программы в памяти формируются списки из тысяч структур, то такой выигрыш уже может быть замечен.

При объявлении битового поля вслед за указанием типа элемента ставится двоеточие (:) и пишется целочисленная константа, задающая ширину поля (т.е. число битов, в которых хранится этот элемент). Ширина поля должна быть целочисленной константой в диапазоне между 0 и заданным общим числом битов, используемых для хранения целого значения типа `int` в вашей системе. Например:

```
struct TPers {
    AnsiString Fam, Nam, Par;
    AnsiString Dep;
    TPers *      pr;
    unsigned     Year : 12;
    bool         Sex : 1;
};
```

Можно задавать неименованное битовое поле. Такое поле используется в структуре как заполнение. Дело в том, что при работе с битовыми полями надо учитывать длину машинного слова. Если следующий элемент структуры не является битовым полем, то место его хранения должно начинаться с нового машинного слова. Для округления объемов памяти до слова, т.е. для заполнения оставшихся неиспользованными битов и вводятся неименованные битовые поля. В приведенном ниже примере неименованное поле шириной в 3 бита используется как заполнение:

```
struct Example {
    unsigned a : 13;
    unsigned : 3;
    unsigned b : 4;
};
```

Можно использовать неименованное битовое поле нулевой ширины, которое воспринимается как указание выравнивать следующее битовое поле по границе нового элемента памяти.

Предупреждение

Манипуляции с битовыми полями являются машинно-зависимыми. Например, в некоторых компьютерах битовые поля могут пересекать границы машинного слова, тогда как в других компьютерах это недопустимо.

13.12 Объединения

Объединение (`union`) — это область памяти, в которой в разные моменты времени могут находиться объекты разных типов. В любой момент времени объединение может содержать максимум один объект, потому что элементы объединения совместно используют одну и ту же область памяти. На программиста возлагается обязанность следить за тем, чтобы к данным в объединении обращались по имени элемента соответствующего типа данных. Если тип ссылки на элемент объединения не соответствует типу данных, хранящемуся в этот момент в объединении, то возникает ошибка, последствия которой зависят от реализации системы.

В разные отрезки времени выполнения программы некоторые объекты могут быть не нужны, т.е. программе требуется только часть ее объектов. Вместо того, чтобы впустую растрачивать память на объекты, которые используются не постоянно, можно поместить их в объединение, где они будут делить между собой одну и ту же область памяти. Число байтов памяти, выделяемых для объединения, должно быть не меньше, чем размер самого большого элемента объединения.

Предупреждение

Не всегда объединение может быть легко перенесено на другие компьютерные платформы. Перенесется ли объединение, или нет, часто зависит от соглашений о выравнивании в памяти типов данных элементов объединения. Так что использование объединений снижает мобильность вашей программы.

Объединения объявляются при помощи ключевого слова **union** в таком же формате, как структуры и классы (см. разделы 13.11.1 и 13.13). Например:

```
union Tunion {  
    int i;  
    double d;  
    char * s;  
};
```

Это объявление создает тип объединения с именем **Tunion**, которое хранит в одной и той же области памяти или целое значение **i**, или действительное значение **d**, или указатель на строку **s**.

Само по себе объявление объединения создает новый тип, но не объект. В дальнейшем для использования объединения надо объявить переменную этого типа, например:

```
Tunion N;
```

К элементам переменной типа объединения можно обращаться так же, как к элементам структуры или класса. Например, вы можете записать операторы:

```
N.i = 5;  
...  
N.d = 5.1;  
...  
char *S = "объединение";  
N.s = S;
```

Но учтите, что при использовании объединения вам надо все время знать, какое значение вы занесли в эту переменную последней операцией присваивания. Если, например, вы выполнили первый из приведенных выше операторов, а затем обратились к элементу **d**, то вы получите бессмысленное значение. А если вы по ошибке обратились в этом случае к элементу **s**, то вас ждут крупные неприятности, поскольку неизвестно, на что будет указывать **s**.

Предупреждение

Использование объединений позволяет экономить ресурсы, но существенно усложняет программирование и затрудняет отладку. Так что решайте, что вам важнее, и не увлекайтесь излишне объединениями.

13.13 Классы

13.13.1 Объявление класса

Класс — это тип данных, определяемый пользователем. То, что в C++Builder имеется множество предопределенных классов, не противоречит этому определению — ведь разработчики C++Builder тоже пользователи C++. Понятия класса, структуры (см. раздел 13.11) и объединения (см. раздел 13.12) в C++ довольно близки друг к другу. Поэтому почти все, что будет далее говориться о классах, применимо также к структурам и объединениям.

Класс должен быть объявлен до того, как будет объявлена хотя бы одна переменная этого класса. Т.е. класс не может объявляться внутри объявления переменной.

Синтаксис объявления класса следующий:

```
class <имя класса> : <список классов - родителей>
{
    public:           // доступно всем
        <данные, методы, свойства, события>
        __published  // видны в Инспекторе Объекта и изменяемы
        <данные, свойства>
    protected:       // доступно только потомкам
        <данные, методы, свойства, события>
    private:          // доступно только в классе
        <данные, методы, свойства, события>
} <список переменных>;
```

Например:

```
class MyClass : public Class1, Class2
{
public:
    MyClass(int = 0);
    void SetA(int);
    int GetA(void);
private:
    int FA;
    double B, C;
protected:
    int F(int);
};
```

Имя класса может быть любым допустимым идентификатором. Идентификаторы классов, наследующих классам библиотеки компонентов C++Builder, приняты начинать с символа «Т».

Класс может наследовать поля (они называются данные-элементы), методы (они называются функции-элементы), свойства, события от других классов — своих предков, может отменять какие-то из этих элементов класса или вводить новые. Если предусматриваются такие классы-предки, то в объявлении класса после его имени ставится двоеточие и затем дается список родителей. В приведенном выше примере предусмотрено множественное наследование классам **Class1** и **Class2**. Если среди классов-предков встречаются классы библиотеки компонентов C++Builder или классы, наследующие им, то множественное наследование запрещено.

Если объявляемый класс не имеет предшественников, то список классов-родителей вместе с предшествующим двоеточием опускается. Например:

```
class MyClass1
{
    ...
};
```

Доступ к объявляемым элементам класса определяется тем, в каком разделе они объявлены. Раздел **public** (открытый) предназначен для объявлений, которые доступны для внешнего использования. Это открытый интерфейс класса. Раздел **published** (публикуемый) содержит открытые свойства, которые появляются в процессе проектирования на странице свойств Инспектора Объектов и которые, следовательно, пользователь может устанавливать в процессе проектирования. Раздел **private** (закрытый) содержит объявления полей и функций, используемых только внутри данного класса. Раздел **protected** (защищенный) содержит объявления, доступные только для потомков объявляемого класса. Как и в случае закрытых элементов, можно скрыть детали реализации защищенных элементов от ко-

нечного пользователя. Однако, в отличие от закрытых, защищенные элементы остаются доступны для программистов, которые захотят производить от этого класса производные классы, причем не требуется, чтобы производные классы объявлялись в этом же модуле.

В приведенном выше примере через объект данного класса можно получить доступ только к функциям **MyClass**, **SetA** и **GetA**. Поля **FA**, **B**, **C** и функция **F** — закрытые элементы. Это вспомогательные данные и функция, которые используют в своей работе открытые функции. Открытая функция **MyClass** с именем, совпадающим с именем класса, это так называемый *конструктор класса*, который должен инициализировать данные в момент создания объекта класса. Присутствие конструктора в объявлении класса не обязательно. При отсутствии конструктора пользователь должен сам позаботиться о задании начальных значений данных — элементам класса.

Перед именами классов-родителей в объявлении класса также может указываться спецификатор доступа (в примере **public**). Смысл этого спецификатора тот же, что и для элементов класса: при наследовании **public** (открытом наследовании) можно обращаться через объект данного класса к методам и свойствам классов-предков, при наследовании **private** подобное обращение невозможно. Подробнее этот вопрос рассмотрен в разделе 13.13.5.

По умолчанию в классах (в отличие от структур) предполагается спецификатор **private**. Поэтому можно включать в объявление класса данные и функции, не указывая спецификатора доступа. Все, что включено в описание до первого спецификатора доступа, считается защищенным. Аналогично, если не указан спецификатор перед списком классов-родителей, предполагается защищенное наследование.

Объявления данных-элементов (полей) выглядят так же, как объявления переменных или объявления полей в структурах:

```
<тип> <имена полей>;
```

В объявлении класса поля запрещается инициализировать. Для инициализации данных служат конструкторы, о которых упоминалось выше и которые рассматриваются подробно в разделе 13.13.4.

Объявления функций-элементов в простейшем случае не отличаются от обычных объявлений функций (см. главу 12 раздел 12.5.1).

После того, как объявлен класс, можно создавать объекты этого класса. Если ваш класс не наследует классам библиотеки компонентов C++Builder, то объект класса создается как любая переменная другого типа простым объявлением. Например, оператор

```
MyClass MC, MC10[10], *Pmc;
```

создает объект **MC** объявленного выше класса **MyClass**, массив **MC10** из десяти объектов данного класса и указатель **Pmc** на объект этого класса.

В момент создания объекта класса, имеющего конструктор, можно инициализировать его данные, перечисляя в скобках после имени объекта значения данных. Например, оператор

```
MyClass MC(3);
```

не только создает объект **MC**, но и задает его полю **FA** значение 3. Если этого не сделать, то в момент создания объекта поле получит значение по умолчанию, указанное в содержащемся в объявлении класса прототипе конструктора.

Создание переменных, использующих класс, можно совместить с объявлением самого класса, размещая их список между закрывающей класс фигурной скобкой и завершающей точкой с запятой. Например:

```
class MyClass : public Class1, Class2
{
    ...
} MC, MC10[10], *Pmc;
```

Если создается динамически размещаемый объект класса (см. главу 12, раздел 12.9), то это делается операцией **new**. Например:

```
MyClass *PMC = new MyClass;
```

или

```
MyClass *PMC1 = new MyClass(3);
```

Эти операторы создают где-то в динамически распределяемой области памяти сами объекты и создают указатели на них — переменные **PMC** и **PMC1**.

Создание объектов класса простым объявлением переменных возможно только в случае, если среди предков вашего класса нет классов библиотеки компонентов C++Builder. Если же такие предки есть, то создание указателя на объект этого класса возможно только операцией **new**. Например, если класс объявлен так:

```
class MyClass2 : public TObject
{
    ...
};
```

то создание указателя на объект этого класса может осуществляться оператором

```
MyClass2 *P2 = new MyClass2;
```

13.13.2 Функции-элементы, дружественные функции, константные функции

Поля данных, исходя из принципа инкапсуляции (см. главу 1 раздел 1.2), всегда должны быть защищены от несанкционированного доступа. Доступ к ним, как правило, должен осуществляться только через функции, включающие методы чтения и записи полей. В этих функциях должна осуществляться проверка данных чтобы не записать случайно в поля неверные данные или чтобы не допустить их неверной трактовки.

Поэтому данные всегда целесообразно объявлять в разделе **private** — закрытом разделе класса. В редких случаях их можно помещать в **protected** — защищенном разделе класса, чтобы возможные потомки данного класса имели к ним доступ.

Хороший стиль программирования

Как правило, делайте данные-элементы класса защищенными, снабжая их при необходимости открытыми функциями чтения и записи. Функции записи позволят вам проверять записываемые данные и обеспечивать тем самым непротиворечивость данных. А функции чтения позволят вам не переписывать программу, даже если вы решили изменить что-то в типе, способах хранения и размещения данных в классе.

Приведем пример. Пусть класс имеет следующее объявление:

```
class MyClass
{
public:
    void SetA(int);    // функция записи
    int GetA(void);    // функция чтения
private:
    int FA;
    double B, C;
};
```

Реализация функций записи и чтения может иметь вид:

```
void MyClass::SetA(int Value)
{
    if(...)          // проверка корректности данных
        FA = Value;
}

int MyClass::GetA(void) {return FA;}
```

В данном случае функция чтения просто возвращает значение поля, но в более сложных классах может потребоваться какая-то предварительная обработка данных. Обратите внимание, что все описания функций-элементов содержат ссылку на класс с помощью операции разрешения области действия (::).

В приведенном примере объявление класса содержит только прототипы функций, а их реализация вынесена из описания класса. Для простых функций реализация может быть размещена непосредственно в объявлении класса. Например:

```
class MyClass
{
public:
    MyClass(int = 0);
    void SetA(int Value) {FA= Value;};    // функция записи
    int GetA(void) {return FA;};         // функция чтения
private:
    int FA;
    double B, C;
};
```

Функции, описание которых содержится непосредственно в объявлении класса, в действительности являются встраиваемыми функциями **inline** (см. главу 12, раздел 12.5.6, в котором обсуждаются достоинства и недостатки таких функций).

Введение описания функций в объявление класса — это плохой стиль программирования: следует избегать смешения открытого интерфейса класса, содержащегося в его объявлении, и реализации класса. Если уж вы хотите реализовать встраиваемые функции, то лучше поместить в объявлении класса их прототип со спецификатором **inline**:

```
inline void SetA(int);    // функция записи
```

и отдельно дать реализацию функции. При этом в реализации спецификатор **inline** не указывается.

Хороший стиль программирования

Объявления классов следует размещать в заголовочном файле модуля, а реализацию функций — элементов в отдельном файле реализации. При этом в объявлении класса должны содержаться только прототипы функций. Это следует из принципа скрытия информации — одного из основных в объектно-ориентированном программировании. Такая организация программы обеспечивает независимость всех модулей, использующих заголовочный файл с объявлением класса, от каких-то изменений в реализации функций-элементов класса.

Функции-элементы класса имеют доступ к любым другим функциям-элементам и к любым данным-элементам, как открытым, так и закрытым. Клиенты класса (какие-то внешние функции, работающие с объектами данного класса) имеют доступ только к открытым функциям-элементам и данным-элементам. Но в некоторых случаях желательно обеспечить доступ к закрытым элементам для функций, не являющихся элементами данного класса. Это можно сделать, объявив соответствующую функцию как *друга класса* с помощью спецификации **friend**. Например, если в объявление класса включить оператор

```
friend void IncFA(MyClass *);
```

то функция **IncFA**, не являясь элементом данного класса, получает доступ к его закрытым элементам. Например, функция **IncFA** может быть описана где-то в программе следующим образом:

```
void IncFA(MyClass *P) {P->FA++;}
```

Дружественными могут быть не только функции, но и целые классы. Например, вы можете поместить в объявление своего класса оператор

```
friend Class1;
```

и все функции-элементы класса **Class1** получают доступ к закрытым элементам вашего класса.

Иногда программист может захотеть создать объект вашего класса как константный с помощью спецификатора **const**. Например:

```
const Class1 MC1(3);
```

Если при этом ваш класс содержит не только функции чтения, но и записи данных, то реакция на такой оператор, введенный пользователем, зависит от версии и настройки компилятора. Компилятор может выдать сообщение об ошибке и отказаться от компиляции, а может просто выдать предупреждение и проигнорировать спецификатор пользователя **const**. Если же ваш класс содержит только функции чтения, то все должно быть нормально. Но компилятор подойдет к этому чисто формально и все равно выдаст предупреждение, а может и отказаться компилировать программу.

Чтобы избежать этого, можно объявить функции чтения как константные. Для этого и в прототипе, и в реализации после закрывающей список параметров круглой скобки надо написать спецификатор **const**. Например, вы можете включить в объявление класса оператор

```
int GetA(void) const;
```

а реализацию этой функции оформить как:

```
int MyClass::GetA(void) const {return FA;}
```

Тогда неприятные замечания компилятора о константных объектах исчезнут.

Хороший стиль программирования

Если предполагается, что объект вашего класса может быть объявлен константным, снабжайте все функции-элементы класса, предназначенные для чтения данных, спецификаторами **const**.

13.13.3 Данные-элементы, статические данные, константные данные

Теперь рассмотрим несколько подробнее данные-элементы. Обычно каждый объект класса имеет свою собственную копию всех данных-элементов класса. Но в определенных случаях во всех объектах класса должна фигурировать только одна копия некоторых данных. Например, это может быть счетчик числа созданных объектов класса.

Единственную копию данных полезно иметь и во многих иных случаях. Например, если в классе имеются некоторые константы, одинаковые для всех объектов класса, то нерационально хранить в каждом объекте собственные копии этих констант. Рациональнее иметь единственные экземпляры этих констант для всех объектов.

Для введения в класс подобных данных используются статические данные, которые содержат информацию «для всего класса». Объявление статических элементов в классе начинается с ключевого слова **static**. Например:

```
static int D;
```

Хороший стиль программирования

Данные, общие для всех объектов класса, надо объявлять как статические данные-элементы. Это сократит затраты памяти и гарантирует единство данных во всех объектах.

Статические элементы могут быть открытыми, закрытыми или защищенными (**protected**). Доступ к открытым статическим элементам класса возможен посредством любого объекта класса или посредством имени класса с помощью бинарной операции разрешения области действия. Например:

```
MyClass::D = 10;
```

Закрытые и защищенные статические элементы класса должны быть доступны открытым функциям-элементам этого класса или друзьям класса.

Статические элементы класса существуют даже тогда, когда не существует никаких объектов этого класса. В этом случае доступ к открытому статическому элементу обеспечивается так же, как указано выше: с помощью имени класса и бинарной операции разрешения области действия. Для обеспечения доступа в отсутствие объектов к закрытому или защищенному элементу класса должна быть предусмотрена открытая статическая функция-элемент, которая должна вызываться с добавлением перед ее именем имени класса и бинарной операции разрешения области действия.

Начальные значения статических элементов (как открытых, так и закрытых) должны задаваться вне объявления класса. Для этого достаточно разместить где-то в файле, например, после объявления класса или среди реализаций функций-элементов (но не внутри их) оператор вида

```
int MyClass::D = 0;
```

Предупреждение

Статическим данным-элементам можно задать начальные значения один и только один раз в области действия файл. Если вы нигде не инициализировали статический элемент данных, будет выдано сообщение компилятора о неразрешенной внешней ссылке и программа не будет скомпилирована. Если вы дважды инициализируете статический элемент, будет выдано сообщение о дублировании инициализации и программа также не будет скомпилирована.

Приведем пример, демонстрирующий все сказанное относительно статических данных-элементов:

```
class MyClass
{
public:
    static int D;
    static int GetD1(void);
    ...

private:
    static int D1;
    ...
};
...
int MyClass::GetD1(void) {return D1;}

int MyClass::D = 0;
int MyClass::D1 = 1;
```

В этом примере имеются два статических элемента данных: открытый **D** и закрытый **D1**. Если пользователь должен иметь возможность получать значения за

крытой статической переменной **D1**, то должна быть предусмотрена функция ее чтения, названная в примере **GetD1**. Она должна быть объявлена открытой (**public**) и статической (со спецификатором **static**). Статической может быть объявлена любая функция, работающая только со статическими данными.

После объявления класса в примере расположена реализация функции **GetD1**. В реализации не требуется указывать спецификатор **static**. Далее приведены предложения, инициализирующие открытые и закрытые статические данные. На этом все, связанное с объявлением и инициализацией статических данных завершается. В дальнейшем вы можете из любой внешней функции обращаться к ним с помощью операции разрешения области действия. Например:

```
i = MyClass::D;  
j = MyClass::GetD1();
```

Среди данных — элементов могут быть объявлены именованные константы. Например:

```
static const int MaxA = 10;  
const int MinA;
```

Значения статических именованных констант могут задаваться в момент их объявления в классе, как показано в предыдущем примере. Инициализация нестатических констант — вопрос более сложный, связанный с построением конструкторов. Он рассматривается в разделе 13.13.4.

13.13.4 Конструкторы и деструкторы

Остановимся теперь на конструкторах класса. Прежде всего отметим, что наличие конструктора в классе не обязательно. Но если конструктор отсутствует, то клиенты класса (внешние функции, использующие класс) должны сами заботиться об инициализации данных, т.е. о задании им некоторых начальных значений. Это не всегда возможно. Например, если класс имеет закрытые данные, предназначенные только для чтения, то для этих данных не предусматриваются открытые функции записи. И клиент не в состоянии присвоить данным какие-то начальные значения.

Конструктором класса называется открытая функция-элемент, которая вызывается в момент создания объекта класса и должна инициализировать данные указанными в вызове значениями или значениями по умолчанию. Конструктор имеет то же имя, что и сам класс.

Пример объявления и реализации конструктора:

```
class MyClass  
{  
public:  
    MyClass(void); // конструктор класса  
    ...  
private:  
    int A;  
  
    ...  
};  
...  
  
MyClass::MyClass(void) {A = 0;}
```

В этом примере объявлен конструктор **MyClass** без параметров, который при создании объекта задает начальное значение поля **A** равным 0. Обратите внимание на то, что в отличие от других функций в объявлении конструктора не указывается тип возвращаемого значения.

Простое задание в конструкторе значений данных в общем случае не гарантирует их целостность. Обычно нужна еще проверка допустимости данных. Например, если в классе есть функция записи **SetA**, осуществляющая такие проверки, то лучше обратиться к ней и при задании начального значения. В этом случае реализация конструктора может иметь вид:

```
MyClass::MyClass(void) { SetA(0); }
```

Создание объекта описанного класса **MyClass** в программе должно осуществляться или объявлением соответствующей переменной:

```
MyClass MC;
```

или динамическим размещением переменной в памяти:

```
MyClass *PMC = new MyClass;
```

В момент выполнения каждого из этих операторов неявным образом вызывается конструктор, устанавливающий начальные значения данных.

Недостатком конструкторов показанного типа является то, что все начальные значения данных задаются в них конструктором. Вызывающая функция никак не может вмешаться в этот процесс и задать какое-то другое значение.

Другой крайностью являются конструкторы, в которых все начальные значения задаются как параметры. Например, прототип конструктора может иметь вид

```
MyClass(int);
```

а его реализация:

```
MyClass::MyClass(int a) { SetA(a); }
```

В этом случае поле **FA** инициализируется параметром, передаваемым в конструктор. Создание объекта подобного класса должно выполняться операторами

```
MyClass MC(1);
```

или

```
MyClass *PMC = new MyClass(1);
```

в которых подразумевается, что начальное значение поля **FA** должно быть равно 1.

Такой конструктор обычно тоже неудобен, поскольку в классе может быть много параметров и задавать значения их всех при создании объекта очень громоздко и чревато ошибками.

Чаще всего используются конструкторы с параметрами по умолчанию (см. главу 12 раздел 12.5.4). В этом случае объявление конструктора может иметь вид:

```
MyClass(int = 0);
```

а его реализация:

```
MyClass::MyClass(int a) { SetA(a); }
```

Объект такого класса можно создавать любым из приведенных ранее операторов создания объекта. Если при создании указывается аргумент, то его значение присваивается полю. Если аргумент не указывается, то присваивается значение по умолчанию (в нашем примере 0). Этот вариант конструктора наиболее гибкий. Поэтому он чаще всего используется при построении классов.

Хороший стиль программирования

Как правило, в классе надо предусматривать конструктор с параметрами по умолчанию.

В объявлении класса могут быть определены не только поля переменных, но и некоторые именованные константы. Например:

```
const int MaxA;
```


Подобная константа может служить, в частности, предельно допустимым значением поля **FA**.

Если такая константа объявлена как статическая (см. раздел 13.13.3), то в ее объявление в классе можно непосредственно включить инициализацию:

```
static const int MaxA = 10;
```

Но тогда это значение клиент при желании не сможет изменить. А задать значение такой константы в конструкторе невозможно, поскольку компилятор не разрешает присваивать значения константам. Выходом из положения является специальный синтаксис конструктора с *инициализатором элементов*. Инициализатор элементов записывается после заголовка конструктора в его реализации, предвзвешивается двоеточием и содержит имена константных данных, после которых в скобках указываются их значения. Например, если в объявлении вашего класса **MyClass** имеются строки

```
const int MaxA;
const int MinA;
```

вводящие две константы — максимальное и минимальное значения переменной **A**, то реализацию конструктора такого класса с описанным ранее прототипом

```
MyClass(int = 0);
```

надо дополнить инициализатором элементов:

```
MyClass::MyClass(int a) : MaxA(10), MinA(1) {SetA(a);};
```

В данном случае инициализатор задает константе **MaxA** начальное значение 10, а константе **MinA** — значение 1.

Можно предоставить пользователю возможность изменять значения констант в момент создания объекта. В этом случае в конструкторе с умолчанием надо предусмотреть для констант соответствующие значения по умолчанию:

```
MyClass(int A = 0, int MaxA = 10, int MinA = 1);
```

или

```
MyClass(int = 0, int = 10, int = 1);
```

(второй вариант менее удобен, так как не позволяет по прототипу функции понять, в какой последовательности должны задаваться параметры).

Тогда реализацию конструктора можно оформить так:

```
MyClass::MyClass(int a, int i, int j) : MaxA(i), MinA(j) { SetA(a); }
```

Создание объектов такого типа может осуществляться, например, такими операторами:

```
MyClass MC;           // умолчание: A = 0, MaxA = 10, MinA = 1
MyClass MC(20);       // задано:   A = 20, MaxA = 10, MinA = 1
MyClass MC(20,15);    // задано:   A = 20, MaxA = 15, MinA = 1
MyClass MC(20,15,2);  // задано:   A = 20, MaxA = 15, MinA = 2
```

Теперь остановимся на деструкторах. Это специальные функции-элементы, срабатывающие при уничтожении динамически размещенного объекта класса и освобождающие занимаемую им память. Имя деструктора совпадает с именем класса, но перед ним записывается символ тильда (~). Как и для конструктора, в деструкторе не указывается возвращаемый тип. Например:

```
class MyClass
{
public:
    ~MyClass(); // деструктор класса
    ...
};
```

Деструкторы необходимы, если конструктор или какие-то функции-элементы класса динамически распределяют память, создавая в ней какие-то объекты. Тогда деструктор должен эти объекты удалять. В остальных случаях можно обычно обойтись без деструктора.

Если деструктор явным образом в классе не объявлен, компилятор сам генерирует необходимые коды освобождения памяти.

13.13.5 Наследование и полиморфизм, виртуальные функции, абстрактные классы

При описании нового класса, производного от какого-то одного или нескольких базовых классов, можно добавлять новые функции-элементы и данные-элементы, сохраняя при этом все элементы родителей, а можно родительские элементы переопределить или перегрузить. В производном классе доступны открытые и защищенные элементы базового класса (прямого или косвенного предшественника). Закрытые элементы базового класса в производном классе недоступны.

Производный класс может наследоваться от базового класса как **public**, **protected** или **private** (см. синтаксис такого наследования в разделе 13.13.1). Защищенное и закрытое наследования встречаются редко и каждое из них нужно использовать с большой осторожностью.

При порождении класса как **public** открытые элементы базового класса становятся открытыми элементами производного класса, а защищенные элементы базового класса становятся защищенными элементами производного класса. Закрытые элементы базового класса никогда не бывают доступны для производного класса.

При защищенном наследовании открытые и защищенные элементы базового класса становятся защищенными элементами производного класса. При закрытом наследовании открытые и защищенные элементы базового класса становятся закрытыми элементами производного класса. При закрытом и защищенном наследованиях несправедливо отношение, что объект производного класса является объектом базового класса.

В целом доступ к элементам базового класса из производного класса можно представить следующей таблицей.

	Тип наследования		
	public открытое наследование	protected защищенное наследование	private закрытое наследование
public	public в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам, дружественным функциям и функциям, не являющимся элементами.	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.	private в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.

Тип наследования			
protected	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.	private в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.
private	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции-элементы базового класса.	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции-элементы базового класса.	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции-элементы базового класса.

Если в классе-наследнике переопределить функцию-элемент (ввести новую функцию с тем же именем), то для объектов этого класса новая функция отменит родительскую. Если обращаться к объекту этого класса, то вызываться будет новая функция. Если все-таки нужно вызвать именно функцию базового класса, надо использовать операцию разрешения области действия.

Пусть, например, вы создали класс форм **Shape**:

```
class Shape
{
public:
    void Draw(void);
    ...
};
```

и наследующий ему класс кругов **Circl**:

```
class Circl : public Shape
{
public:
    void Draw(void);
};
```

в каждом из классов объявили метод рисования **Draw**, а затем в программе выполнили операторы

```
Shape *PQ1 = new Shape;
PQ1->Draw();           // вызов Draw класса Shape
Circl *PQ2 = new Circl;
PQ2->Draw();           // вызов Draw класса Circl
PQ2->Shape::Draw();    // вызов Draw класса Shape
((Shape *)PQ2)->Draw(); // вызов Draw класса Shape
```

В комментариях к коду указано, функции **Draw** каких классов вызывают эти операторы. Как видно, если мы обращаемся через указатель к самому объекту типа **Circl**, то вызывается переопределенная в нем функция. Но если мы обращаемся к нему как к объекту базового класса (последний из приведенных операторов)

или соответствующим образом используем операцию разрешения области, то вызывается функция базового класса.

Если бы в классе-наследнике **Circl** отсутствовала функция **Draw**, то все приведенные операторы вызывали бы функцию базового класса.

Таким образом, механизм наследования позволит использовать функции базового класса или переопределять их.

Теперь рассмотрим другую задачу. Пусть мы имеем несколько классов, наследующих **Shape**: **Circl** (круг), **Rectangle** (прямоугольник), **Square** (квадрат) и т.п. Каждый из этих классов имеет свою функцию **Draw**, которая умеет рисовать соответствующую форму. Мы хотим работать с объектами этих фигур, как с объектами базового класса **Shape**, не разбираясь в истинной природе каждого объекта. И при этом хотим, чтобы программа сама понимала, что это за объект и как его рисовать. Например, мы хотим создать массив указателей на объекты различных форм:

```
Shape *ShapeArray[10];
```

загрузить его указателями на объекты разных фигур:

```
ShapeArray[0] = new Circl;  
ShapeArray[1] = new Rectang;  
ShapeArray[2] = new Square;  
...
```

и затем в цикле выполнять рисование этих фигур:

```
for(int i = 0; i < 3; i++)  
    ShapeArray[i]->Draw();
```

Рассмотренный ранее механизм наследования такую задачу решить не может. Поскольку ко всем объектам мы обращаемся через тип их базового класса **Shape**, то только функция этого класса и будет вызываться.

Поставленную задачу *полиморфизма* позволяют решить виртуальные функции. Они не связаны с другими функциями с тем же именем в классах — наследниках. Если в классах — наследниках эти функции переопределены, то при обращении к такой функции во время выполнения будет вызываться та из виртуальных функций с одинаковыми именами, которая соответствует классу объекта, указанного при вызове. Поэтому, если в базовом классе **Shape** объявить функцию **Draw** как виртуальную, то задача будет решена и каждая фигура будет рисоваться своей функцией.

Синтаксически это оформляется следующим образом. В базовом классе **Shape** функция объявляется следующим образом:

```
virtual void Draw(void);
```

И это все! Если функция была однажды объявлена виртуальной, она остается виртуальной и во всех классах наследниках. Таким образом для решения задачи полиморфизма хватило одного спецификатора **virtual**. Правда, обычно предпочитают для большей ясности программы в классах-наследниках тоже вводить спецификатор **virtual**, чтобы была ясна суть этих функций для тех, кто будет строить наследников данного класса. Но с точки зрения языка C++ это не обязательно.

Иногда в базовом классе определяют *чистую виртуальную функцию* (абстрактную функцию). Это функция, для которой не указана реализация. Для того, чтобы определить такую функцию, достаточно указать, что ее тело равно нулю:

```
virtual void Draw(void)=0;
```

В нашем примере именно так целесообразно объявить функцию **Draw** в базовом классе **Shape**, поскольку непонятно, как можно нарисовать просто абстрактную фигуру. Реализация для чистой полиморфной функции не пишется.

Класс, в котором имеется хоть одна чистая виртуальная функция, называется *абстрактным*. Для абстрактного класса невозможно создать объект. Такие классы предназначены только для построения на их основе конкретных классов-наследников.

13.13.6 Особенности классов, наследующих классам библиотеки компонентов C++Builder

13.13.6.1 Свойства

О некоторых особенностях построения классов, наследующих классам библиотеки компонентов C++Builder, уже говорилось ранее. К этим особенностям относится невозможность для таких классов множественного наследования и необходимость создавать объекты только с помощью операции **new**. Теперь остановимся на других особенностях, связанных с понятиями *свойства* и *события*.

Понятие свойства (**property**), объединяет поле данных и функции (методы) его записи и чтения. В рассматриваемых классах сами поля объявляются как обычно, но, как правило, в разделе **private**. Традиционно идентификаторы полей совпадают с именами соответствующих свойств, но с добавлением в качестве префикса символа 'F'.

Свойство объявляется оператором вида:

```
__property <тип> <имя> = {read=<имя поля или метода чтения>
                           write=<имя поля или метода записи>
                           <директивы запоминания>
                           и значения по умолчанию};
```

Если в разделах **read** или **write** этого объявления записано имя поля, значит предполагается прямое чтение или запись данных.

Если в разделе **read** записано имя метода чтения, то чтение будет осуществляться только функцией с этим именем. Функция чтения — это функция без параметра, возвращающее значение того типа, который объявлен для свойства. Имя функции чтения принято начинать с префикса **Get**, после которого следует имя свойства.

Если в разделе **write** записано имя метода записи, то запись будет осуществляться только процедурой с этим именем. Процедура записи — это процедура с одним параметром того типа, который объявлен для свойства. Имя процедуры записи принято начинать с префикса **Set**, после которого следует имя свойства.

Если раздел **write** отсутствует в объявлении свойства, значит это свойство только для чтения и пользователь не может задавать его значение.

Директивы запоминания определяют, как надо сохранять значения свойств при сохранении пользователем файла формы **.dfm**. Чаще всего используется директива

```
default <значение по умолчанию>
```

Она не задает начальное значение. Это дело конструктора. Директива просто говорит, что если пользователь в процессе проектирования не изменил значение свойства по умолчанию, то сохранять значение свойства не надо.

Приведем пример. Пусть требуется объявить класс с именем **MyClass**, наследующий непосредственно **TObject** и имеющий свойство целого типа с именем **A**. Тогда объявление этого класса может иметь вид:

```
class MyClass1 : public TObject
{
private:
    int FA;
protected:
    void __fastcall SetA(int); // функция записи
published:
    __property int A = {read = FA, write = SetA, default = true};
};
```

Здесь вводится закрытое поле **FA**, объявляется защищенная функция **SetA**, используемая для записи значения в это поле, и вводится опубликованное свой-

во **A**, оперирующее этим полем. В объявлении свойства после ключевого слова **read** записано просто имя поля. Это означает, что функция чтения отсутствует и пользователь может читать непосредственно значение поля. После ключевого слова **write** следует ссылка на функцию записи **SetA**, с помощью которой будут записываться в поле **A** новые значения. В этой функции можно предусмотреть какие-то проверки допустимости вводимого значения **A**.

Описание этой функции может иметь вид:

```
void __fastcall MyClass1::SetA(int Value)
{
    if(...) FA = Value;
}
```

В приведенном примере описание свойства **A** помещено в раздел **published**. Следовательно, если этот класс описывает создаваемый вами новый компонент, то после его установки в систему свойство **A** будет появляться в окне Инспектора Объектов при использовании этого компонента. Если перенести объявление свойства в раздел **public**, то свойством по-прежнему можно будет пользоваться, но только во время выполнения приложения, поскольку в окне Инспектора Объектов оно появляться не будет. Если удалить из определения свойства слово **write** с последующей ссылкой на функцию записи, то свойство станет свойством только для чтения, т.к. изменить его непосредственно будет невозможно.

Для свойств типа массивов приведенный ранее оператор **__property** изменяется следующим образом:

```
__property <тип> <имя> <список размерностей> =
    { read=<имя поля или метода чтения>
      write=<имя поля или метода записи>
      <директивы запоминания
      и значения по умолчанию>;
```

Список размерностей представляет собой последовательность квадратных скобок, в которых записывается тип размерности и может записываться идентификатор. Приведем в качестве примера возможный вариант описания класса матриц действительных чисел размером **N x M**:

```
// Класс матриц действительных чисел
class Matrix
{
    float *data;
    int N;    // число строк
    int M;    // число столбцов
public:
    Matrix(int,int);
    ~Matrix() { delete[ ] data; }
    __property float Items [int i] [int j] =
        { read=GetItems, write=SetItems };
private:
    float __fastcall GetItems(int i, int j);
    void __fastcall SetItems(int i, int j, float value);
};

Matrix::Matrix(int n, int m)    // конструктор
{
    data = new float [n*m];      // создание экземпляра класса
    for(int i = 0; i < n*m; i++) // инициализация
        data[i] = 0.;
    N = n;
    M = m;
}
```

```

void __fastcall Matrix::SetItems(int i, int j, float value)
{
    // запись значения value в элемент (i,j)
    if ((i<1) || (i>N) || (j<1) || (j>M))
        ShowMessage("Недопустимые индексы (" + IntToStr(i) +
            ", " + IntToStr(j) + ")");
    else data[(i-1) * M + j - 1] = value;
}

float __fastcall Matrix::GetItems(int i, int j)
{
    // чтение значения элемента (i,j)
    if ((i<1) || (i>N) || (j<1) || (j>M))
        ShowMessage("Недопустимые индексы (" + IntToStr(i) +
            ", " + IntToStr(j) + ")");
    else return data[(i-1) * M + j - 1];
}

```

В приведенном коде создается класс матриц **Matrix**. Класс имеет открытое свойство **Items**, к которому можно обращаться как к двумерному массиву. Об этом говорит его определение в операторе **__property: float Items [int i] [int j]**. Указание в списке размерностей идентификаторов **i** и **j** не является обязательным. Список мог бы иметь вид: **[int] [int]**.

Задание размерностей изменяет вид функций чтения и записи. В функцию чтения **GetItems** передаются два целых параметра, определяющих индексы читаемого элемента матрицы. В приведенном примере индексы матриц отсчитываются от 1, а не от нуля, что, вероятно, более удобно пользователю. В функцию записи **SetItems** помимо записываемого значения **value** также передаются индексы того элемента, в который должно быть записано это значение.

Создание экземпляра матрицы в программе может, осуществляться, например, оператором:

```
Matrix x(4,5);
```

Этот оператор создает матрицу **x** размерностью 4×5. Запись и чтение элементов матрицы в программе осуществляется через свойство **Items**. Например:

```
x.Items[2][3] = 1.5;
float y = x.Items[2][3];
```

Первый из этих операторов заносит значение 1,5 в 3-ий элемент 2-ой строки, а второй оператор читает это значение.

Дополнительные примеры описания свойств при создании нового класса см. в главе 7 в разделе 7.3.3.

13.13.6.2 События

Событие — это специальное свойство, являющееся указателем функции. В C++Builder тип обобщенного указателя на функцию, которой передается один параметр типа **TObject** (обычно **this**), — **TNotifyEvent**. Подобный тип используется в C++Builder для событий типа **OnClick** и многих других, которые передают в обработчик только один параметр — **TObject *Sender**. Если требуется ввести в класс подобное событие, достаточно определить в объявлении класса соответствующее поле и метод работы с ним. Например:

```

private:
    ...
    TNotifyEvent FMyEvent;
    ...
__published:
    ...
    __property TNotifyEvent MyEvent = {read= FMyEvent,write= FMyEvent};

```


Остается только вызвать в нужный момент обработчик событий пользователя, если пользователь его предусмотрел. Проверка, имеется ли обработчик пользователя, осуществляется проверкой соответствующего события как булевой величины, возвращающей **true**, если пользователь предусмотрел свой обработчик. Значит, при возникновении события надо проверять, имеется ли обработчик пользователя, и, если имеется, то вызывать его. Для этого можно использовать оператор вида:

```
if(FMyEvent) OnMyEvent(this);
```

Функция **OnMyEvent**, которая вызывается этим оператором, это и есть обработчик пользователя. Его имя совпадает с именем свойства, перед которым добавляется префикс «On».

Место, куда надо включать подобный оператор, зависит от вида события. Если событие вызывается каким-то из ваших методов, то вызов обработчика пользователя надо осуществлять из этого метода. Если событие связано с какими-то сообщениями, поступающими от других приложений или от Windows, то надо предусмотреть обработчик соответствующего сообщения (это подробно рассмотрено в главе 6 в разделе 6.3.3) и из него вызывать обработчик пользователя.

Если в обработчик события надо передать какие-то параметры помимо **this**, то тип функции **TNotifyEvent** уже не подходит и надо объявить свой собственный тип. Это объявление делается с помощью ключевого слова **__closure**. Например:

```
typedef void __fastcall (__closure *TMyEvent)
    (System::TObject *Sender, bool& MyParam);

class T : public TObject
{
private
    TMyEvent FMyEvent;
public
    published
        __property TMyEvent FMyEvent = {read= FMyEvent, write= FMyEvent};
    ...
}
```

Примеры объявления и использования событий вы можете посмотреть в главе 7 в разделе 7.3.5.

Выше было рассмотрено введение в класс какого-то нового события. Если же вам надо переопределить одно из традиционных событий, связанных с клавиатурой, мышью и т.п., то это можно сделать, переопределив соответствующий стандартный обработчик родительского класса.

13.13.7 Шаблоны классов

C++ позволяет определять шаблоны классов, называемые также родовыми (generic) классами или генераторами классов. Иногда их называют параметризованными типами, так как они имеют один или большее количество параметров типа, определяющих настройку шаблона класса на специфический тип данных при создании объекта класса.

Для того, чтобы использовать шаблонные классы, программисту достаточно один раз описать шаблон класса. Каждый раз, когда требуется реализация класса для нового типа данных, программист, используя простую краткую запись, сообщает об этом компилятору, который и создает исходный код для требуемого класса.

Шаблоны классов задаются аналогично шаблонам функций (см. раздел 12.5.8 главы 12). Описание шаблона отличается от описания класса первой строкой

```
template <class идентификатор> class имя класса
```

В этой строке **идентификатор** является произвольным именем формального типа, который используется далее в описании шаблона. Например:

```
template <class T> class Matrix
{
...
};
```

Этот заголовок объявляет о создании шаблона класса **Matrix** и задает идентификатор **T** для формального типа данных. Этот идентификатор следует использовать в описании класса вместо указания типа соответствующих данных. Приведем в качестве примера шаблон класса матриц, аналогичных классу, описанному в разделе 13.13.6.1.

```
// Шаблон класса матриц
template <class T> class Matrix
{
    T *data;
    int N;    // число строк
    int M;    // число столбцов
public:
    Matrix(int,int);
    ~Matrix() { delete[] data; }
    __property T Item {int i} [int j] =
        { read=GetItem, write=SetItem };
private:
    T __fastcall GetItem(int i, int j);
    void __fastcall SetItem(int i, int j, T value);
};

// конструктор
template <class T> Matrix      ::Matrix(int n, int m)
{
    data = new T[n*m];
    for(int i = 0; i < n*m; i++)
        data[i] = 0;
    N = n;
    M = m;
}

template <class T> void __fastcall
    Matrix::SetItem(int i, int j, T value)
{
    // запись значения value в элемент (i,j)
    if((i<1)|| (i>N)|| (j<1)|| (j>M))
        ShowMessage("Недопустимые индексы (" + IntToStr(i) +
            "," + IntToStr(j) + ")");
    else data[(i - 1) * M + j - 1] = value;
}

template <class T> T __fastcall
    Matrix      ::GetItem(int i, int j)
{
    // чтение значения элемента (i,j)
    if((i<1)|| (i>N)|| (j<1)|| (j>M))
        ShowMessage("Недопустимые индексы (" + IntToStr(i) +
            "," + IntToStr(j) + ")");
    else return data[(i - 1) * M + j - 1];
}
```

Если вы сравните этот код с приведенным ранее в разделе 13.13.6.1, то увидите, что основное отличие заключается в замене типа **float**, который использовался в разделе 13.13.6.1, на формальный тип **T**. Благодаря этому в самом шаблоне не указывается действительный тип хранимых данных. И при создании конкретного

экземпляра класса можно будет задавать любой тип: целый, действительный, комплексный и т.п. Другое отличие приведенного кода от рассмотренного в разделе 13.13.6.1 заключается в форме ссылок заголовков элементов-функций на шаблон класса.

Создание экземпляра матрицы конкретного типа в программе может, осуществляться, например, оператором:

```
Matrix<float> x(4,5);
```

Этот оператор создает матрицу *x* действительных чисел размерностью 4 x 5. Отличие от приведенного в разделе 13.13.6.1 аналогичного оператора заключается в том, что после имени класса в угловых скобках указывается тип, для которого создается экземпляр класса. Компилятор заменит на этот тип (в данном случае **float**) формальный тип **T**, использованный в описании шаблона.

Запись и чтение элементов матрицы в программе осуществляется точно так же, как в разделе 13.13.5.1, через свойство **Items**. Например:

```
x.Items[2][3] = 1.5;  
float y = x.Items[2][3];
```

Первый из этих операторов заносит значение 1,5 в 3-ий элемент 2-ой строки, а второй оператор читает это значение.

Глава 14

Справочные данные по интегрированной среде разработки C++Builder

14.1 Структура меню C++Builder 5

14.1.1 Меню файлов File

Меню файлов используется для открытия, сохранения, закрытия и печати новых или существующих проектов и файлов, а также для добавления новых форм и модулей в открытый проект. Меню имеет следующие разделы.

New — создать новый объект из элементов Депозитария

Создаваемый объект может быть любым элементом Депозитария (хранилища) объектов, включая новый проект. После выбора этого раздела меню появляется диалоговое окно New Items, которое позволяет выбрать вид элемента начиная от окна и кончая сервером Web. Вы можете также воспользоваться услугами различных Мастеров (Wizard) по созданию проектов и форм. Вид окна New Items вы можете увидеть на ряде рисунков в главе 2 и в главе 7. Там же обсуждаются вопросы работы с этим окном и заимствования проектов и форм из Депозитария.

Окно New Items имеет четыре предопределенных страницы, содержание которых вы не можете изменять:

New	Страница стандартных заготовок проектов и их элементов
ActiveX	Страница активных форм и компонентов ActiveX
Multitier	Страница компонентов для CORBA (Common Object Request Broker Architecture — стандарта построения приложений с распределенными объектами), MTS (Microsoft transaction server) и автоматных серверов
Your project	Страница, на которую автоматически заносятся все формы вашего текущего проекта

Содержание остальных страниц может изменяться пользователем (см. раздел 7.4 главы 7). При поставке C++Builder в Депозитарии имеются дополнительные страницы:

Forms	Примеры различных форм, которые можно использовать как заготовки для многих ваших приложений
Dialogs	Диалоговые окна различного назначения
Data Modules	Модули данных

Projects	Проекты различного вида, которые могут служить прототипами для ваших разработок
Business	Мастера, облегчающие разработку ряда форм специального назначения

Основные элементы страницы New:

Application	Создание нового приложения (файлов .cpp , .h и .bpr)
Batch file	Создание командного файла .bat
C File	Создание программы на языке C. Если выбрать этот элемент, не открывая проект, то создастся проект программы на C
Component	Создание нового компонента
Console Wizard	Создание консольного приложения, работающего в окне DOS и использующего Win32
Control Panel Application	Создание приложения, размещающего свою пиктограмму в окне «Панель управления»
Control Panel Module	Создание нового модуля приложения, созданного ранее элементом Control Panel Application
CPP File	Создание программы на языке C++. Если выбрать этот элемент, не открывая проект, то создастся проект программы на C++
Data Module	Создание и включение в проект нового модуля данных для связи с базой данных. Позволяет графически документировать и визуально проектировать структуру модуля
DLL Wizard	Создание новой библиотеки DLL
Form	Создание и включение в проект новой формы
Frame	Создание и включение в проект нового фрейма
Header File	Создание нового заголовочного файла .h
Library	Создание библиотеки
MFC Wizard	Создание проекта, совместимого с Microsoft foundation classes (MFC)
Package	Создание нового пакета
Project Group	Создание новой группы проектов
Report	Создание и включение в проект формы для разработки отчетов QuickReport. Имеется также элемент QuickReport Wizard на странице Business, который позволяет решать аналогичную задачу в диалоге.
Resource DLL Wizard	Мастер DLL ресурсов, используемый при интернационализации приложения
Text	Создание текстового документа ASCII
Unit	Создание и включение в проект нового модуля без формы
Web Server Application	Создание нового сервера Web

New Application — создать новый проект

Если у вас в данный момент не открыт ни один проект или текущее состояние открытого проекта уже сохранено на диске, C++Builder закроет текущий проект и создаст новый. Это подразумевает создание новой пустой формы, нового файла ее модуля, соответствующего заголовочного файла и файла проекта. Если имеется не закрытый проект, то предварительно будет задан вопрос о его сохранении.

По умолчанию новый проект создается с пустой формой. Но при настройке Депозитария (см. раздел 14.2.3) вы можете изменить это, задав в качестве проекта по умолчанию какой-то из проектов, хранящихся в Депозитарии.

New Form — создать новую форму

По умолчанию будет создана новая пустая форма, в отличие от специальных форм, имеющих в Депозитарии объектов. Но при настройке Депозитария (см. раздел 14.2.3) вы можете изменить это, задав в качестве формы по умолчанию какую-то форму из хранящихся в Депозитарии.

New Frame — создать новый фрейм

Данная команда создает и включает в текущий проект новый фрейм (см. раздел 3.7.8).

Open — открыть существующий модуль, проект или текстовый файл

В этом разделе можно открыть модуль, проект или текстовый файл. Если открывается модуль, то он появляется в окне Редактора Кода, становится видима связанная с ним форма, но в проект этот модуль не включается. Это можно использовать для просмотра каких-то модулей из других проектов, в частности, из предоставляемых с C++Builder многочисленных примеров. Аналогично загружается в Редактор Кода открываемый текстовый файл. Закреть открытый в окне Редактора Кода модуль или текстовый файл можно щелчком правой кнопки мыши и выбором команды Close Page. При открытии проекта потребуются закрыть и при желании сохранить ранее открытый проект.

Open Project — открыть существующий проект или группу

Этот раздел позволяет открыть существующий проект или группу проектов. Если вы открываете проект (файл .bpr), а в этот момент у вас открыт другой проект, то текущий проект закроется и откроется новый. То же произойдет, если вы откроете группу проектов (файл .bpg). Если у вас открыта группа проектов, то открытие нового проекта заменит проекты, содержащиеся ранее в группе.

Reopen — открыть проект или модуль, с которыми работали раньше

Этот раздел является списком нескольких последних проектов или файлов, которые вы открывали ранее. Чтобы открыть файл, содержащийся в этом списке, достаточно просто выделить его в списке. Это проще и быстрее, чем пользоваться разделами меню File | Open. Правда, еще проще воспользоваться соответствующей быстрой кнопкой (см. таблицу 2.1 в разделе 2.2.3).

Save — сохранить текущий модуль

На диске сохраняется текущий модуль, с которым вы работаете в окне Редактора Кода. Если ранее этот модуль уже был сохранен, то команда Save сохраняет его без дополнительных запросов под тем же именем. Если же он еще не сохранялся на диске, то команды Save работает как команда Save As, предлагая вам стандартный диалог Windows для задания имени сохраняемого файла.

Сохранение модуля означает не только сохранение файла **.crr**, но и всех файлов связанной с ним формы, ресурсов и т.п.

Save As — сохранить текущий модуль под новым именем

При выборе этого раздела открывается стандартное диалоговое окно, которое позволяет сохранить текущий модуль под новым именем. Вы можете захотеть сделать это, если собираетесь вносить радикальные изменения во фрагмент кода. Это позволяет вам сохранить изменения и при необходимости вернуться к прежнему коду, если вы сделали ошибку в новом. Вы можете также для этих целей использовать Borland TeamSource (см. раздел 2.4.4.2), если эта система включена в вашу версию C++Builder.

Save Project As — сохранить текущий проект под новым именем

При выборе этого раздела меню открывается стандартное диалоговое окно, которое позволяет сохранить текущий проект под новым именем. Это позволяет сохранить файл проекта под новым именем для какого-то последующего его использования. Для модулей, входящих в проект, имена файлов не изменяются.

Save All — сохранить текущий проект и все его файлы

Выбор этого раздела меню сохраняет все, что в данный момент открыто — проект, группу проектов и все прочие файлы: модули, ресурсы и т.п.

Close — закрыть текущий модуль или проект

Выбор этого раздела закрывает файл, связанный с активным окном. Если активным было окно Редактора Кода с загруженным в него модулем, то закрывается этот модуль и связанная с ним форма. Если же в окне Редактора Кода был выделен файл проекта или активным было окно Менеджера Проектов, то закрывается текущий проект вместе со всеми его файлами. Если вы не сохранили ваш модуль или проект в его текущем состоянии, C++Builder спросит, хотите ли вы сохранить внесенные изменения.

Close All — закрыть все файлы текущего проекта или группы

Выбор этого раздела закрывает все файлы текущего проекта или группы. Если вы не сохранили какие-то из файлов в их текущем состоянии, C++Builder спросит, хотите ли вы сохранить внесенные изменения.

Include Unit Hdr — связать активный модуль с другим модулем проекта

Выбор этого раздела меню доступен только в проектах с несколькими модулями. При выборе раздела открывается диалоговое окно, в котором вы можете выбрать модуль, с которым хотите связать активный в данный момент модуль. В результате в текущий модуль будет вставлена директива **#include**, подключающая заголовочный файл указанного вами модуля. Тогда из текущего модуля можно будет обращаться к открытым объектам, методам, функциям, переменным указанного модуля. Конечно, вы можете и без этого раздела меню вставить в код соответствующую директиву. Этот раздел меню просто облегчает эту операцию.

Print — напечатать текст модуля или форму

Выбрав этот раздел меню, вы можете распечатать выбранный элемент проекта. Если вы выделили в C++Builder форму и выполнили команды File | Print, то появ-

вится диалоговое окно печати формы (Print Form). В этом диалоговом окне вы можете выбрать, как бы вы хотели напечатать форму. Имеются опции:

Proportional	Делается попытка напечатать изображение формы того же размера, который виден на экране. При этом используется свойство формы PixelsPerInch — число пикселей на дюйм. В зависимости от значения этого свойства форма может оказаться занимающей более одной страницы
Print to fit page	Увеличивает или уменьшает размер изображения, подгоняя его под размер страницы, заданный при установке принтера. Пропорции формы сохраняются
No scaling	Масштабирование не используется. Размер изображения может изменяться в зависимости от используемого принтера и может занимать более одной страницы

Если вы хотите напечатать программный код, то должны выделить окно с этим кодом, выполнить команду File | Print и на экране появится диалоговое окно печати текста (Print Selection). В нем вы можете задать опции, определяющие формат печати: Print selected block — печать только выделенного текста (предполагается, что вы выделили фрагмент текста), Header/Page Number — печать вверху каждой страницы ее номера и имени файла, Line Numbers — печать слева от каждой строки ее номера, Syntax Print — печать выделений в окне редактирования (жирный шрифт, курсив, подчеркивание), Use Color — печатать в цвете (на цветном принтере), Wrap lines — переносить строки (в противном случае строки, не помещающиеся на странице, усекаются), Left Margin — левое поле, измеряемое числом символов.

Exit — выход

Выбор этого раздела меню приводит к выходу из ИСР C++Builder. Если ваш проект не сохранен в текущем состоянии, C++Builder предложит вам сохранить его перед уходом.

14.1.2 Меню редактирования Edit

Раздел главного меню Edit включает меню редактирования для работы с текстом и компонентами в процессе проектирования. Меню имеет следующие разделы.

Undo/Undelete — отменить предыдущую операцию

Этот раздел появляется в меню в виде команды Undelete (восстановить удаленное) или Undo (отменить предыдущее действие) в зависимости от того, какая команда выполнялась перед этим. Если вы только что удалили объект или какой-то код клавишей Delete или с помощью команды Delete в меню Edit, то данный раздел будет иметь название Undelete. Он позволит вам восстановить то, что вы непосредственно перед этим удалили. А если вы только что добавили в проект код, раздел будет называться Undo. Это позволит вам отменить последнее добавление.

Redo — отменить предыдущую команду Undo

Команда Redo противоположна команде Undo. Redo возвращает вас назад к состоянию, которое было до выполнения команды Undo или даже до выполнения ряда последовательных команд Undo.

Cut — вырезать в буфер обмена компоненты или фрагмент текста

Эта команда вырезает выделенные элементы (компонент, группу компонентов на форме или фрагмент текста) и помещает их в буфер Clipboard. Выделенные элементы удаляются из текущей формы или текста. Эта команда в совокупности с командой Paste позволяет, в частности, переносить группу компонентов из одного контейнера (например, панели) в другой (другую панель) — см. раздел 2.5.2 главы 2.

Copy — копировать в буфер обмена компоненты или фрагмент текста

Эта команда копирует выделенные элементы (компонент, группу компонентов на форме или фрагмент текста) и помещает их в буфер Clipboard. Выделенные элементы не удаляются из текущей формы или текста. Эта команда в совокупности с командой Paste позволяет, в частности, копировать группу компонентов из одного контейнера или формы в другой контейнер или форму, которая может находиться в другом проекте. В совокупности с командой Paste позволяет также копировать фрагменты текста из других проектов или примеров, поставляемых с C++Builder.

Paste — прочитывать содержимое буфера обмена в форму или в текст

Эта команда копирует содержимое Clipboard в текущую форму или в текст. Совместно с командами Cut и Copy может использоваться для перемещения и копирования компонентов, групп компонентов и фрагментов кода.

Delete — удалить компоненты или фрагмент текста

Команда удаляет выделенные на форме или в тексте элементы. Если вы ошиблись, то имеется возможность восстановить ошибочно удаленные элементы с помощью команды Undo. Помните, что удаленные выделенные на форме или в тексте элементы не помещаются в буфер Clipboard.

Select All — выделить все

Эта команда выделяет все компоненты на текущей форме или весь код в текущем модуле в зависимости от того, с чем вы в данный момент работаете.

Align to Grid — выравнивать по сетке

Команда выравнивает размещенные на форме компоненты по узлам сетки. Правда, если вы установили опцию Snap to Grid на странице Preference с помощью команды Tools | Environment Options, то раздел меню Align to Grid не нужен. Все компоненты будут автоматически размещаться на форме в узлах сетки.

Bring to Front — переместить наверх

Команда помещает выделенный компонент сверху всех остальных (на верх Z-последовательности, но с учетом того, что все неоконные компоненты располагаются позади оконных). Это полезно, если вы разместили на форме ряд компонентов и они накладываются друг на друга. В этом случае вы можете решить, что какой-то скрытый компонент, лежащий под другими, должен быть наверху. Для этого надо выделить этот компонент и затем выполнить данную команду меню.

Send to Back — переместить вниз

Эта команда противоположна по смыслу команде Bring to Front. Она помещает выбранный компонент или компоненты позади всех других компонентов (в низ Z-последовательности, но с учетом того, что все не оконные компоненты располагаются позади оконных).

Align — выровнять местоположение компонентов в группе

При выборе этого раздела меню появляется диалоговое окно Alignment (Выравнивание). Опции этого окна позволяют выбрать ряд вариантов выравнивания компонентов на форме по горизонтали и вертикали (см. раздел 2.5.2.5).

Size — выровнять размеры группы компонентов

Выбор этого раздела меню позволяет изменять размеры компонента до заданных значений ширины и высоты. Если вы выделили несколько компонентов, данный раздел позволяет увеличить размеры всех этих компонентов по горизонтали, вертикали или в обоих направлениях до размеров наибольшего из выделенных компонентов на странице или сократить их размеры до размера наименьшего из них (см. раздел 2.5.2.5).

Scale — масштабировать все компоненты на форме

Используя команду Scale, вы можете пропорционально изменить масштаб всего расположенного на форме. Все размеры можно увеличивать или уменьшать вплоть до ста раз. В появляющемся диалоговом окне вам надо задать Scaling factor — масштабирующий коэффициент в %. Например, задав 200 вы увеличите все компоненты в 2 раза.

Tab Order — установить последовательность табуляции

C++Builder позволяет установить последовательность смены активных компонентов, расположенных в данном контейнере — на форме, панели и т.п., при нажатии пользователем клавиши табуляции Tab. Команда Tab Order высвечивает на экране диалоговое окно редактирования последовательности табуляции (Edit Tab Order). В окне помещен список имен всех компонентов, размещенных в данном контейнере. Вы можете изменить их последовательность, выделяя соответствующий элемент и нажимая кнопки со стрелками «вверх» или «вниз». Это удобнее, чем устанавливать свойства каждого компонента вручную.

Creation Order — установить последовательность создания невизуальных компонентов

Команда позволяет управлять последовательностью, в которой создаются невизуальные компоненты. Эта последовательность может быть важна, если одни из этих компонентов используют свойства других, полагая, что те существуют и инициализированы. Если эти компоненты не создаются в правильной последовательности, то обращение к несуществующему компоненту вызовет генерацию исключения.

Flip Children — зеркальное отображение размещения

Раздел позволяет зеркально преобразовать размещение (справа налево) компонентов формы или конкретного контейнера (панели), выбрав из подменю один из двух вариантов: All — все компоненты формы (при этом внутри каждого контейнера тоже происходит зеркальное отображение) или Selected — дочерние компоненты выделенного контейнера. Чаще всего это используется при разработке вариантов приложений, предназначенных для стран Востока.

Lock Controls — зафиксировать компоненты

После того, как вы разместили и выровняли компоненты, их местоположение полезно зафиксировать этой командой. Иначе в процессе последующей работы над проектом вы можете случайно сдвинуть тот или иной компонент, когда будете его выделять курсором, и всю работу по выравниванию придется начинать заново.

Команда Lock Controls зафиксировывает расположение всех компонентов на форме и не позволит их перемещать. Если в дальнейшем у вас все-таки возникнет потребность изменить расположение компонентов, то выполните повторно команду Edit | Lock Controls, и компоненты будут разблокированы.

Add to Interface — добавить в интерфейс компонента ActiveX

С помощью этого раздела вы можете добавить новый метод, событие или свойство в интерфейс компонента ActiveX.

CORBA Refresh — обновить классы CORBA

Команда обновляет классы CORBA (Common Object Request Broker Architecture — стандарт построения приложений с распределенными объектами), чтобы они отражали изменения в файлах IDL вашего проекта.

Use CORBA Object — использовать объект CORBA

Команда автоматически генерирует код для связи с объектом CORBA (Common Object Request Broker Architecture — стандарт построения приложений с распределенными объектами).

14.1.3 Меню поиска Search

Выпадающее меню Search используется для локализации текста, объектов, модулей, переменных и символов в Редакторе Кода. Меню содержит следующие разделы.

Find — найти

C++Builder выполняет поиск первого вхождения заданной последовательности символов в тексте. При выполнении команды на экране появляется диалоговое окно поиска текста Find Text, предоставляя в распоряжение пользователя ряд опций: учет регистра (Case sensitivity), поиск заданной последовательности символов только как целого слова (Whole words only), направление поиска (Direction), область поиска (Scope) — во всем файле (Global) или только в выделенном фрагменте (Selected text), от курсора (From cursor) или в заданной области независимо от положения курсора (Entire scope).

Find in Files — найти в файлах

Выбор раздела меню Find in Files (поиск в файлах) позволяет вам проводить поиск заданного текста и просматривать каждое его вхождение в нижней части окна Редактора Кода. Наличие списка всех вхождений делает во многих случаях эту команду более удобной, чем Find. Опции диалогового окна позволяют вам проводить поиск во всех файлах текущего проекта (Search all files in project), во всех открытых файлах (Search all open files) и во всех файлах заданного каталога (Search in directories). Остальные опции поиска аналогичны команде Find.

Replace — заменить

Диалоговое окно этой команды подобно окну команды Find, но содержит окошко для ввода заменяемой последовательности символов — Text to find, и окошко Replace with, в котором вы задаете текст замены.

Search Again — искать снова

Эта команда повторяет последний проведенный вами поиск, который вы задавали в диалоговом окне Find Text.

Incremental Search — быстрый поиск по вводимым символам

Выбрав этот раздел меню, начните печатать какое-нибудь слово. C++Builder* найдет в тексте первое вхождение печатаемой вами последовательности символов. Это прекрасный инструмент в тех случаях, когда вы только приблизительно знаете, что именно хотите найти.

Go to Line Number — перейти на строку с заданным номером

Этот раздел меню позволяет вам ввести номер строки (в пределах числа строк, имеющих в файле вашего приложения), на которую вы хотите перейти.

Go to Address — перейти по заданному адресу

Этот раздел меню доступен только во время отладки. Он позволяет в соответствующем диалоговом окне указать адрес команды в памяти, по которому вы хотите перейти. Это может быть, например, адрес, связанный с последней ошибкой в приложении. После указания адреса вы попадете в окно CPU на строку, соответствующую указанному адресу.

14.1.4 Меню просмотра View

Выпадающее меню просмотра View позволяет вывести на экран или скрыть различные элементы среды проектирования C++Builder и открыть окна, связанные с интегрированным отладчиком. Меню содержит следующие разделы.

Project Manager — Менеджер Проектов

Эта команда активизирует окно Менеджера Проектов — Project Manager (см. раздел 2.4.3).

Translation Manager — Менеджер Трансляции

Эта команда активизирует окно Менеджера Трансляции, используемое при интернационализации приложений (см. раздел 4.7.1.2).

Object Inspector — Инспектор Объектов

Эта команда активизирует Инспектор Объектов — Object Inspector.

To-Do List — список планируемых задач проектирования

Эта команда отображает окно списка To-Do List, содержащего список планируемых задач проектирования (см. раздел 2.4.4.1).

Alignment Palette — палитра выравнивания

Эта команда активизирует очень удобный инструмент — палитру выравнивания Alignment Palette (см. раздел 2.5.2.5). Это визуальный вариант диалогового окна выравнивания Alignment, которое вызывается командой меню Edit | Align.

ClassExplorer — окно Исследователя Классов

Этот раздел меню активизирует окно Исследователя Классов, помогающего анализировать текст модуля и вносить элементы в объявления классов (см. раздел 2.5.3.2).

Component List — список компонентов

Выбор этого раздела меню отобразит диалоговое окно алфавитного списка всех компонентов в библиотеке. Кнопка Add to Form позволяет разместить выбранный компонент на форме.

Window List — список открытых окон

Выбор этого раздела меню откроет диалоговое окно со списком всех открытых окон C++Builder. Вы можете выбрать в нем нужное вам окно, и оно будет активизировано. Этот раздел очень полезен, если у вас открыто много окон, перекрывающих друг друга, и вы «потеряли» нужное вам окно.

Debug Windows — вспомогательное меню отладки

Этот раздел имеет подменю, включающее в себя разделы:

Breakpoints	Активизирует диалоговое окно списка точек прерывания — Breakpoint List. Оно показывает все заданные для отладки точки прерывания. Если вы щелкнете правой кнопкой мыши в этом окне, появится меню, позволяющее вам добавлять, изменять и удалять отладочные точки прерывания, формулировать условия прерывания (см. раздел 2.6.7)
Call Stack	Активизирует диалоговое окно стека вызовов — Call Stack. Оно показывает последовательность, в которой вызываются процедуры и функции в выполняемом приложении
Watches	Активизирует диалоговое окно наблюдения. С помощью этого окна можно наблюдать в процессе отладки программы заданное множество переменных или выражений, содержащих эти переменные (см. раздел 2.6.4)
Local Variables	Показывает значения локальных переменных выполняемой функции
Threads	Показывает список текущих процессов, которые в данный момент выполняются. Так как Windows 95/98 и NT являются многозадачными системами, вы можете организовать в своем приложении несколько ветвей, которые будут выполняться независимо
Modules	Активизирует диалоговое окно списка модулей — Module List. Окно содержит перечень всех модулей, загруженных в память при выполнении данного проекта. Этот список включает выполняемые модули и библиотеки DLL
Event Log	Активизирует окно, отображающее произошедшие события (см. раздел 2.6.9)
CodeGuard Log	Отображает ошибки, обнаруженные CodeGuard (если CodeGuard у вас установлен)
CPU	Активизирует окно CPU, позволяющее наблюдать процесс выполнения приложения на уровне ассемблера
FPU	Активизирует окно FPU, позволяющее наблюдать информацию Floating-Point Unit в CPU или MMX — расширенную версию Intel для процессоров Pentium

Desktops — управление конфигурациями окон

Выбор конфигурации окон и установка конфигурации окон в режиме отладки (см. раздел 2.2.9).

Toggle Form/Unit — переключение между формой и кодом модуля

Данная команда переключает вас между формой и ее модулем.

Units — список модулей проекта

Выбор этого раздела меню вызовет диалоговое окно, которое покажет список всех модулей (units) вашего проекта. Вы можете щелкнуть на том модуле, который хотите посмотреть, и он появится в окне Редактора Кода.

Forms — список форм проекта

Выбор этого раздела меню вызовет диалоговое окно, которое покажет список всех форм вашего проекта. Вы можете щелкнуть на имени той формы, которую хотите посмотреть, и она активизируется на экране.

Type Library — библиотека типов

Используется при преобразовании компонентов C++Builder в элементы ActiveX, серверы автоматизации, объекты MTS и COM. Библиотека типов — Type Library является составным документом OLE, содержащим информацию о типах данных, функциях-элементах и классах объектов, предоставленную управляющими активными элементами ActiveX или серверами. Когда выделяется Type Library в панели списка объектов — Object List Panel, выполнение команды Type Library делает доступными страницу атрибутов и страницу используемых модулей Uses.

Страница атрибутов содержит информацию о текущей выбранной библиотеке. Когда Type Library выделена в главной панели списка объектов, то на странице атрибутов присутствуют следующие атрибуты и флаги: Name, GUID, Version, LCID, HelpFile, Help String, Help Context и др.

New Edit Window — новое окно Редактора Кода

Выбор этого раздела меню открывает новое окно Редактора Кода, оставляя на месте прежнее. Текущий модуль вашего окна редактирования появится и в новом окне. Наличие на экране нескольких окон редактирования позволяет одновременно видеть коды двух модулей.

Toolbars — панель быстрых кнопок

Эта команда вызывает подменю, позволяющее выбрать состав инструментальной панели быстрых кнопок (см. раздел 14.2.1).

14.1.5 Меню проекта Project

Раздел меню Project используется для компиляции и построения приложения. Меню содержит следующие разделы.

Add To Project — добавить в проект

Выбор этого раздела меню позволяет вам добавить в проект существующий модуль и связанную с ним форму. Когда вы добавляете модуль в проект, C++Builder автоматически добавляет в файл проекта .bpr соответствующую директиву #include.

Remove From Project — удалить из проекта

Выбор этого раздела меню позволяет вам удалить из проекта существующий модуль и связанную с ним форму. Когда вы удаляете модуль из проекта, C++Builder автоматически удаляет из файла проекта соответствующую директиву `#include`.

Import Type Library — импортировать в проект библиотеку типов

Выбор этого раздела меню открывает диалоговое окно, позволяющее импортировать в проект одну из зарегистрированных в системе библиотек типов.

Add To Repository — добавить текущую форму, фрейм или проект в Депозитарий

Этот раздел меню добавляет текущую форму, фрейм или проект в Депозитарий объектов — Object Repository. Это позволяет использовать добавленный элемент повторно, что сокращает время на разработку новых проектов.

View Source — смотреть исходный файл проекта

Эта команда заносит в окно Редактора Кода головной файл проекта с функцией `WinMain` или `main`.

Languages — языки

Эта команда позволяет добавлять, удалять и обновлять DLL ресурсов, используемые при интернационализации проектов (см. раздел 4.7.1.2), а также позволяет выбирать язык в процессе отладки интернационализированных приложений.

Edit Option Source — открыть файл проекта

Открывает в Редакторе Кода файл проекта `.bpr`. Этот файл содержит информацию об опциях проекта, о версии, об используемых пакетах и т.п.

Add New Project — добавить новый проект в группу

Эта команда создает выбором из Депозитария новый проект и добавляет его в текущую группу проектов.

Add Existing Project — добавить ранее созданный проект в группу

Эта команда открывает один из имеющихся проектов и добавляет его в текущую группу проектов.

Compile Unit — компилировать

Эта команда осуществляет компиляцию только того модуля, который выделен вами в окне Редактора Кода или в Менеджере Проектов. Команда позволяет наиболее быстро проверить наличие ошибок или замечаний при компиляции модуля, так как не осуществляется компоновка программы и не компилируются никакие другие модули. Если компиляция прошла успешно, создается объектный файл `.obj` откомпилированного модуля.

Make project — скомпоновать проект

Эта команда выполняет компиляцию всех тех модулей, тексты которых были изменены с момента предыдущей компоновки проекта (см. раздел 2.6.1). Если компиляция прошла успешно, то создаются объектные файлы модулей `.obj` и осу-

существляется компоновка программы. Если и она прошла успешно, то создается выполняемый модуль `.exe`.

Build project — компилировать весь проект

Эта команда компилирует все модули вне зависимости от того, изменялось ли в них что-нибудь с момента последней компиляции вашей программы (см. раздел 2.6.1). Этим данная команда отличается от `Make project`. Если компиляция прошла успешно, то создаются новые объектные файлы всех модулей `.obj` и осуществляется компоновка программы. Если и она прошла успешно, то создается выполняемый модуль `.exe`.

Information for project — информация о проекте

Эта команда открывает окно, в котором содержится информация о компиляции проекта: число строк в кодах (Source Compiled), объем кода в байтах (Code Size), объем памяти в байтах, необходимый для хранения данных (Data Size), объем памяти в байтах, необходимый для хранения локальных переменных (Initial Stack Size), общий объем выполняемого модуля в байтах (File size) и информацию об успешности последней компиляции.

Make All Projects — скомпоновать все проекты группы

Эта команда осуществляет компиляцию всех файлов проектов в группе, которые были изменены с момента последней компиляции (см. раздел 2.6.1). Если компиляция прошла успешно, то создаются объектные файлы модулей `.obj` и осуществляется компоновка программ. Если и она прошла успешно, то создаются выполняемые модули `.exe`.

Build All Projects — компилировать все файлы всех проектов

Эта команда осуществляет компиляцию всех файлов всех проектов в группе независимо от того, изменялись они или нет (см. раздел 2.6.1). Компиляция осуществляется в той последовательности, в которой проекты отображены в окне Менеджера Проектов. Изменить при желании эту последовательность можно, выделив проект в окне Менеджера Проектов, щелкнув правой кнопкой мыши и выбрав команду `Build Sooner` (построить раньше) или `Build Later` (построить позднее).

Web Deployment Options — сделать установки для развертывания формы или элемента ActiveX на сервере Web

Эта команда позволяет сделать необходимые установки для развертывания формы или элемента ActiveX на сервере Web.

Web Deploy — развертывание формы или элемента ActiveX на сервере Web

Когда вы закончили проектирование активной формы, выполните команду `Web Deploy`, чтобы развернуть ее на сервере Web.

Options — опции проекта

Этот раздел меню активизирует диалоговое окно опций проекта — `Project Options`, в котором вы можете установить опции компиляции, компоновки и задать каталоги.

14.1.6 Меню выполнения Run

Раздел меню Run содержит выпадающее меню с командами, обеспечивающими выполнение и отладку вашей программы. Меню содержит следующие разделы.

Run — компилировать и выполнить проект

Данный раздел меню запускает выполнение вашего приложения C++Builder. Если до этого не была осуществлена компиляция программы в ее текущем состоянии, то перед запуском эта компиляция выполняется. Если вы работаете с группой проектов, то команда относится к активному в данный момент проекту (см. раздел 2.4.3).

Attach to Process — подключиться к отладке процесса

Данный раздел меню открывает окно, в котором перечислены все процессы (программы), выполняемые на компьютере в данный момент. Выбрав процесс и сделав на нем двойной щелчок или нажав кнопку Attach, вы можете подключиться к нему (появится соответствующее окно CPU).

Parameters — параметры командной строки

Этот раздел меню вызывает диалоговое окно, которое позволяет задать параметры командной строки, необходимые при запуске приложения, или указать хост (ведущее приложение) при отладке DLL, или указать компьютер при удаленной отладке.

Register ActiveX Server — зарегистрировать в Windows активный сервер ActiveX

Этот раздел меню доступен, если текущий проект — ActiveX. Данный раздел дает возможность зарегистрировать ваш активный сервер ActiveX в реестре Windows 95/98. Если вы проведете подобную регистрацию, то ваш управляющий активный элемент можно будет вызывать, применяя программу просмотра Web или другие приложения. Прежде, чем использовать управляющий элемент в первый раз, его надо зарегистрировать.

Unregister ActiveX Server — снять с регистрации в Windows активный сервер ActiveX

Этот раздел меню доступен, если текущий проект — ActiveX. Данный раздел дает вам возможность снять ваш активный сервер ActiveX с регистрации в реестре Windows 95/98. Тем самым экземпляр вашего активного элемента удаляется из системы.

Install MTS Objects — установка объекта MTS

Эту команду можно использовать, если на вашем компьютере установлен Microsoft transaction server (MTS) и ваш проект является проектом MTS. Тогда эта команда устанавливает ваш объект MTS в пакет MTS.

Install COM+ Objects — установить объекты в приложение COM+

Эту команду можно использовать, если ваша система поддерживает COM+. Тогда эта команда вызывает диалоговое окно, позволяющее установить объекты текущего приложения в приложение COM+.

Step Over — выполнить по шагам без захода в функции

Эта команда вызывает пошаговое выполнение приложения, по одному оператору за шаг, причем любая функция выполняется, как если бы она была одним оператором программы. Это удобно, если вы хотите посмотреть поведение вашей программы, не заходя внутрь каждой функции (см. раздел 2.6.6).

Trace Into — выполнить по шагам с заходом в функции

Эта команда вызывает пошаговое выполнение приложения, по одному оператору за шаг, но любая функция выполняется тоже в пошаговом режиме (см. раздел 2.6.6).

Trace to Next Source Line — выполнить до следующей команды

Эта команда позволяет выполнить приложение до следующей выполняемой команды. Ее удобно применять при работе с окном CPU.

Run to Cursor — выполнить до курсора

Эта команда выполняет ваше приложение вплоть до той точки в исходном тексте, где находится курсор (см. раздел 2.6.6).

Run Until Return — выполнить до выхода из функции

Эта команда выполняет ваше приложение до выхода из текущей функции и останавливается на операторе, следующем за вызовом этой функции.

Show Execution Point — показать точку выполнения

Выполните эту команду, если вы закрыли окно редактирования и находитесь в процессе пошаговой отладки программы. Это вернет вас в окно редактирования, причем курсор будет расположен на операторе, который будет выполняться следующим.

Program Pause — пауза в выполнении

Эта команда вызывает паузу в выполнении приложения и вы можете спокойно поработать с окном Watch.

Program Reset — завершение приложения

Эта команда прекращает выполнение вашего приложения и выгружает его из памяти.

Inspect — открыть окно Инспектора Отладки

Команда доступна только во время выполнения приложения при останове средствами отладки или вследствие генерации исключения. При останове вы можете поставить курсор на имя интересующей вас переменной, или компонента, или функции и выполнить команду Run | Inspect. Инспектор Отладки позволяет исследовать различные данные: переменные, массивы, классы, функции, указатели. Причем, не только исследовать, но и изменять значения переменных и свойств компонентов (см. раздел 2.6.8).

Evaluate/Modify — изменение значения переменной при выполнении

Эта команда позволяет вам с помощью диалогового окна Evaluate/Modify (см. раздел 2.6.5) изменять значение переменной в процессе выполнения программы.

Более того, вы можете написать некоторое выражение, включающее переменные вашего приложения, и это выражение будет немедленно посчитано.

Add Watch — добавить переменную в окно наблюдения

Этот раздел дает вам один из способов добавить наблюдаемую переменную в список Watch (см. раздел 2.6.4). Другой способ — выполнить View | Watches, сделать щелчок правой кнопкой мыши на окне Watch List и выбрать раздел Add в выпадающем меню.

Add Breakpoint — добавить точку прерывания

Эта команда предоставляет вам один из способов добавить новую точку в список точек прерывания или изменить существующую точку (см. раздел 2.6.7). Другой способ сделать то же самое — выполнить View | Breakpoint, сделать щелчок правой кнопкой мыши в окне Breakpoint List и выбрать раздел Add в выпадающем меню.

14.1.7 Меню компонентов Component

Раздел Component главного меню содержит выпадающее меню, которое позволяет работать с компонентами: создавать новые компоненты, изменять палитру компонентов и т.п. Меню содержит следующие разделы.

New Component — создать новый компонент

Этот раздел меню активизирует диалоговое окно, которое помогает создавать новые компоненты C++Builder.

Install Component — установить новый компонент в пакет

Этот раздел меню позволяет вам установить новый компонент C++Builder в новый или уже существующий пакет C++Builder.

Import ActiveX Library — импортировать элемент ActiveX в пакет

Этот раздел меню позволяет импортировать активный управляющий элемент ActiveX, который уже зарегистрирован в системе, в новый или уже существующий пакет C++Builder.

Create Component Template — создать шаблон компонента

Этот раздел меню становится активным, когда вы выделили на форме компонент или совокупность компонентов. Он позволяет вам выделить, например, Table и DataSource, и затем скомбинировать их в один компонент, который может рассматриваться на форме как единое целое и может быть помещен на какой-либо странице палитры компонентов.

Install Packages — показать пакеты

Этот раздел меню позволяет просмотреть список всех имеющихся пакетов и указать, какие пакеты должны компилироваться в ваше приложение (см. раздел 7.6). Вы можете видеть также текущий список компонентов каждого пакета, редактировать состав пакетов.

Configure Palette — конфигурировать палитру компонентов

Данная команда позволяет добавлять и удалять компоненты, видимые на экране в палитре компонентов (см. раздел 14.2.2).

14.1.8 Меню баз данных Database

Раздел меню Database содержит команды, позволяющие создавать, модифицировать и просматривать ваши базы данных. Меню содержит следующие разделы.

Explore — вызвать SQL Explorer

Этот раздел меню вызывает инструмент исследования баз данных — SQL Explorer. Он позволяет просматривать и редактировать структуры баз данных.

SQL Monitor — вызвать SQL Monitor

Этот раздел меню вызывает программу SQL Monitor. Монитор позволяет наблюдать прохождение запросов SQL и то, как они обрабатываются в вашем приложении.

Form wizard — вызвать мастера разработки форм с базами данных

Мастер форм баз данных поможет вам создать форму для работы с базами данных. Он откроет базу данных, с которой вы хотите связаться, и поможет вам спроектировать экран для данных, содержащихся в файлах.

14.1.9 Меню инструментов Tools

Раздел Tools главного меню C++Builder предоставляет возможность просматривать и изменять установки среды C++Builder, модифицировать список программ в подменю Tools. Это частично формируемый вами раздел меню ИСП, в состав которого вы можете изменить исходя из наиболее часто решаемых вами задач. Вы можете вставить в это меню и какие-то свои собственные программы. По умолчанию меню содержит следующие разделы.

Environment Options — опции окружения

Этот раздел меню выводит диалоговое окно настройки окружения — Environment Options. В этом окне вы можете изменять установки многих опций.

Editor Options — опции Редактора Кода

Этот раздел меню выводит диалоговое окно настройки Редактора Кода (см. раздел 14.2.5).

Debugger Options — опции отладчика

Этот раздел меню выводит диалоговое окно настройки отладчика — Debugger Options, в котором вы можете изменять установки, связанные с отладкой приложений (см. раздел 14.2.8).

Translation ToolsOptions — опции интернационализации

Этот раздел меню выводит диалоговое окно настройки инструментария Integrated Translation Environment (ITE), используемого при интернационализации проектов (см. раздел 4.7.1.2).

Repository — Депозитарий объектов

Этот раздел меню вызывает диалоговое окно просмотра Депозитария объектов — Object Repository (см. раздел 14.2.3). Оно позволяет просмотреть объекты, которые вы поместили в Депозитарий командой Project | Add To Repository. Вы можете также добавлять и удалять объекты из Депозитария.

Translation Repository — депозитарий переводов

Этот раздел меню вызывает диалоговое окно депозитария переводов, хранящего переводы для интернационализации проектов (см. раздел 4.7.1.2).

Visibroker SmartAgent — запуск и останов SmartAgent

Раздел позволяет запустить или остановить Visibroker Smart Agent. Smart Agent должен быть запущен для клиентского приложения CORBA, чтобы связаться с сервером приложений CORBA.

IDL Repository — регистрация файла IDL в проекте с Interface Repository CORBA

Раздел позволяет вызвать диалоговое окно Update IDL Repository, в котором вы можете зарегистрировать свой файл интерфейса IDL.

Configure Tool — настройка разделов меню Tools

Этот раздел меню позволяет вам настроить C++Builder, добавляя инструменты в меню Tools. Это обеспечивает большую гибкость в настройке среды C++Builder под ваши задачи (о настройке меню Tools см. в разделе 14.2.4).

Database Desktop — вызов Database Desktop

Раздел вызывает программу Database Desktop, позволяющую создавать и редактировать базы данных Paradox, dBASE, SQL и др.

TeamSource — система управления проектами

Раздел вызывает главное окно системы Borland TeamSource, организующей управление большими проектами (см. раздел 2.4.4.2). Этот раздел присутствует не во всех версиях C++Builder 5.

Package Collection Editor — создание и редактирование собрания пакетов

Раздел вызывает Редактор Собрания Пакетов, позволяющий создавать новые и редактировать существующие собрания пакетов.

Visual C++Project Conversion Utility — преобразование проектов Visual C++

Раздел преобразует проекты Microsoft Visual C++ 5.0 и 6.0 (файлы .dsp и .dsw) в соответствующие файлы проекта C++Builder.

Image Editor — вызов редактора изображений

Раздел вызывает редактор изображений — битовых матриц, пиктограмм и т.п.

14.1.10 Меню справки Help

Разделы меню Help позволяют работать со справочной системой C++Builder. Меню содержит разделы C++Builder Help — вызов справки по C++Builder и C++, раздел C++Builder Tools — вызов справок по инструментарию C++Builder 5, Windows API/SDK Help — вызов справок по Windows, ряд разделов получения информации через Интернет, а также раздел Customize, позволяющих настраивать справочную систему (см. раздел 2.5.3.4).

14.2 Настройка Интегрированной Среды Разработки C++Builder

14.2.1 Настройка инструментальной панели

Инструментальная панель содержит быстрые кнопки, дублирующие команды меню и существенно облегчающие работу. Благодаря технологии Drag&Dac, вы можете осуществлять перестроение инструментальных панелей, просто перетаскивая их мышью (см. раздел 2.2.8). Но возможности настройки панелей гораздо шире. Вы можете делать какие-то из панелей невидимыми и можете изменять наборы быстрых кнопок панелей.

Наиболее простой способ реорганизации панелей — сделать те из них, которые вам сейчас не нужны, невидимыми. Для этого выполните команду View | Toolbars. Можно обойтись и без этой команды, просто щелкнув правой кнопкой мыши на одной из панелей. В обоих случаях вы увидите меню, содержащее индикаторы отдельных панелей: Standard, View, Debug, Custom, Component Palette, Desktops, CORBA. Можете выключить какие-то из этих индикаторов (например, индикатор Custom, соответствующий панели, которая по умолчанию содержит только кнопку справки). Освободившееся место можно использовать под расширение каких-то других панелей. В дальнейшем в любой момент с помощью того же меню вы можете опять восстановить установку индикатора и панель станет видимой.

В том же меню, всплывающем при щелчке на панели правой кнопкой мыши, имеется раздел Customize, который позволяет настроить состав отдельных панелей, добавляя или удаляя какие-то быстрые кнопки. При выборе этого раздела появится диалоговое окно, представленное на рис. 14.1. В этом окне страница Toolbars позволяет управлять видимостью панелей, а расположенная на этой странице кнопка Reset восстанавливает вид панели по умолчанию. Если вы выберете в окне одну из панелей и нажмете Reset, то после запроса о подтверждении уничтожения всех изменений панели она примет вид, который имела по умолчанию. Таким образом, если вы как-то неудачно настроили панель, у вас всегда есть возможность вернуть ее к тому состоянию, которое продумали за вас создатели C++Builder.

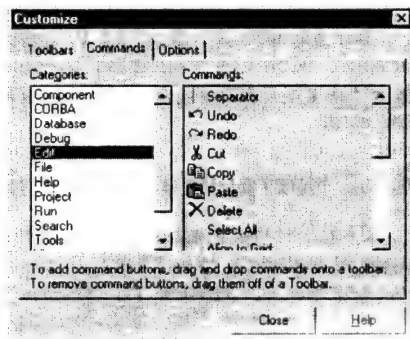
На странице Commands окна рис. 14.1 вы можете выбрать меню (Categories) и его раздел (Commands), для которого вы хотите добавить кнопку на панель. Далее можно мышью перетащить выбранную кнопку на панель. Чтобы удалить кнопку с инструментальной панели, надо перетащить ее оттуда мышью.

На странице Options окна рис. 14.1 вы можете включить или выключить опции Show tooltips — показ ярлычков быстрых кнопок при задержке над ними курсора мыши, и Show shortcut keys on tooltips — показ на этих ярлычках соответствующих сочетаний «горячих» клавиш.

Находясь на любой странице диалогового окна рис. 14.1 вы можете переупорядочивать кнопки любой инструментальной панели, просто перетаскивая их мышью.

Рис. 14.1.

Окно настройки инструментальных панелей

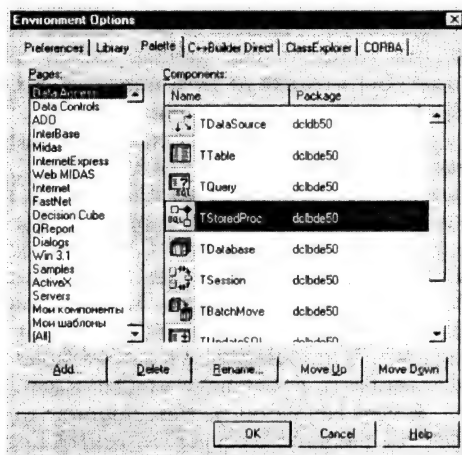


14.2.2 Настройка палитры компонентов

Вызвать настройку палитры компонентов можно щелчком правой кнопки мыши на палитре и выбором команды Properties из всплывшего меню. Можно выполнить для этого команду Component | Configure Palette. Можно также выполнить команду Tools | Environment Options и перейти в открывшемся диалоговом окне на страницу Palette (рис. 14.2).

Рис. 14.2.

Страница настройки палитры компонентов в окне Environment Options



Опции окна позволяют работать со страницами палитры. Для этого надо перейти в окно Pages и нажать кнопку Add, чтобы добавить новую страницу (на рис. 14.2 вы видите внизу две новые страницы — Мои компоненты и Мои шаблоны), кнопку Rename, чтобы переименовать страницу, кнопку Delete, чтобы удалить страницу (она должна быть к этому моменту пустой), кнопки Move Up или Move Down, чтобы изменить последовательность страниц в палитре. Впрочем, последовательность проще изменять, просто перетаскивая мышью в левой панели страницу на новое место в списке.

Последний раздел All показывает список компонентов всех страниц. Если выделить этот раздел, то вид правого окна Components несколько изменится. В нем появляется три столбца: Name, Package и Page. Щелкнув на одном из этих столбцов вы можете упорядочить список компонентов соответственно по имени компонента, по пакету, в который он включен, или по страницам библиотеки. При выделенном разделе All в окне появляется кнопка Default. Если щелкнуть на ней, восстановится последовательность и состав страниц, принятый в C++Builder по умолчанию.

Перейдя в окно Components вы можете изменять состав страниц, перетаскивая мышью компонент с одной страницы на другую или делая кнопкой Hide какие-то компоненты невидимыми (кнопка Hide появляется на месте кнопки Delete, которую вы видите на рис. 14.2, при выделении какого-нибудь компонента в правой панели). Можете изменять последовательность компонентов на странице кнопками Move Up и Move Down, или просто перетаскивая соответствующую строку вверх или вниз.

14.2.3 Настройка Депозитария

Депозитарий является хранилищем форм, фреймов, проектов. Вы можете реорганизовывать страницы Депозитария, вносить на них свои формы и проекты (см. раздел 7.4 главы 7), создавать новые страницы. Для реорганизации страниц щелкните в окне Депозитария правой кнопкой мыши и выберите из контекстного

меню раздел Properties. Вы попадете в окно реорганизации Депозитария, представленное на рис. 14.3. В это же окно можно попасть, выполнив команду Tools | Repository.

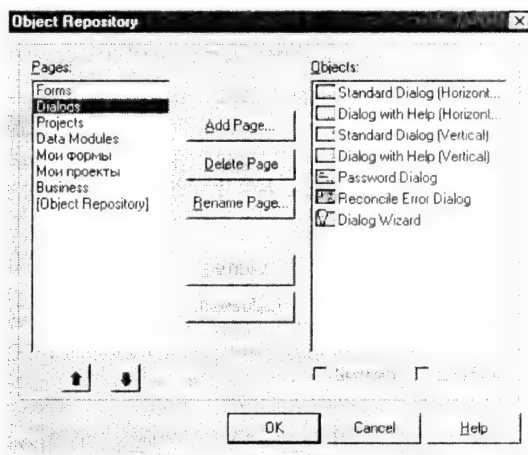
В левом списке Pages (страницы) этого окна вы видите список тех страниц Депозитария, которые вы можете перестраивать. Выделив строку в левом списке вы увидите в правом списке Objects объекты, отображенные на этой странице. Кнопка Edit Object позволяет отредактировать информацию об объекте, кнопка Delete Object позволяет удалить объект из Депозитария. Мышью вы можете перетащить объект из правого окна на другую страницу Депозитария, отображенную в левом окне. Если вы перетащите объект в строку [Object Repository], то этот компонент не будет представлен ни на одной странице окна, хотя по-прежнему будет храниться в Депозитарии. Впоследствии вы можете его извлечь из [Object Repository] и перенести мышью на любую страницу.

Кнопка Add Page позволяет добавить новую страницу в Депозитарий (см. на рис. 14.3 добавленные страницы Мои формы и Мои проекты), кнопка Delete Page удаляет пустую страницу, кнопка Rename Page позволяет переименовать закладку страницы. Кнопки со стрелками позволяют переместить выделенную страницу вверх или вниз. Впрочем, проще перетащить мышью строку выделенной страницы на нужную позицию.

Если вы выделите в правом окне одну из форм, то вам будут доступны два индикатора (см. рис. 14.3): New Form — сделать данную форму формой по умолчанию при добавлении новой формы в проект (при выполнении команды File | New Form), и Main Form — сделать данную форму главной формой по умолчанию (она будет автоматически включаться в создаваемый новый проект вместо пустой формы).

Рис. 14.3.

Окно реорганизации Депозитария



Если вы выделите в правом окне один из проектов, то вам будет доступен индикатор New Project — открывающий данный проект при выполнении вами в последующем команды File | New Application.

Таким образом, реорганизация Депозитария позволяет вам настраивать некоторые разделы меню File.

14.2.4 Настройка меню Tools

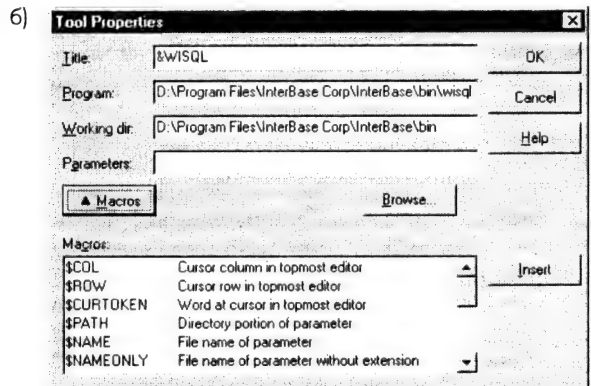
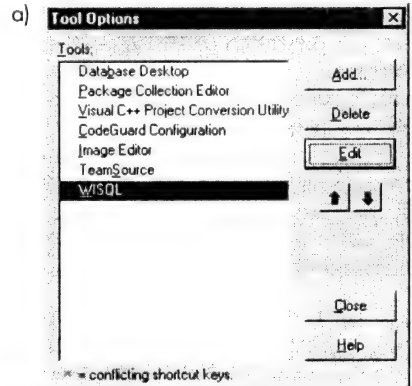
Меню среды разработки Tools может быть настроено добавлением в него разделов, вызывающих те или иные приложения. Таким образом вы можете расширить возможности главного меню C++Builder, приспособив его для своих задач. Для настройки надо выполнить команду Tools | Configure Tool. Перед вами откроется диало-

говое окно, представленное на рис. 14.4 а. В нем содержится список тех разделов меню Tools, которые вы можете удалить (кнопка Delete), переставлять (кнопки со стрелками), редактировать (кнопка Edit). Разделы, входящие в меню безусловно, в этом списке не отражены.

Кнопка Add позволяет добавить в меню новые разделы. При нажатии этой кнопки открывается диалоговое окно, представленное на рис. 14.4 б (на рисунке оно показано с развернутым списком макросов). Для ввода нового раздела меню вы должны в окошке Program написать имя выполняемой программы вместе с путем к нему. В окошке Working Dir вы указываете рабочий каталог выполняемой программы. Оба эти окошка проще всего заполнять, указывая программу в дереве каталогов компьютера с помощью кнопки Browse. Выбор программы этой кнопкой автоматически заполняет оба окошка и вам остается только, если необходимо, подправить рабочий каталог.

Рис. 14.4.

Окно конфигурирования меню Tools (а) и окно ввода нового раздела (б) с развернутым списком макросов



В окошке Title вы должны написать название вводимого вами раздела меню. При этом можете использовать амперсant для указания клавиши ускоренного доступа, как это обычно делается в меню.

Перечисленные выше окна вы должны заполнить. В окно Parameters вы можете занести строку параметров, которую хотите передать в вызываемую программу. При записи этой строки вы можете использовать макросы, список которых разворачивается кнопкой Macros (на рис. 14.4 б этот список развернут). Вы можете использовать следующие макросы:

\$COL	Расширяется до номера позиции курсора в активном окне Редактора Кода. Например, если позиция курсора 50, то C++Builder передаст в запускаемую программу параметр «50»
\$ROW	Расширяется до номера строки курсора в активном окне Редактора Кода. Например, если курсор находится в строке 10, то C++Builder передаст в запускаемую программу параметр «10»
\$CURTOKEN	Расширяется до слова (лексемы), в котором расположен курсор в активном окне Редактора Кода. Например, если курсор находится в слове Label , то C++Builder передаст в запускаемую программу параметр «Label»
\$PATH	Расширяется до каталога файла, указанного в макросе как параметр. Сам макрос записывается в строку как \$PATH(). В скобках вы можете указать файл. Например, макрос \$PATH(\$EDNAME) передаст в запускаемую программу каталог файла, расположенного в активном окне Редактора Кода
\$NAME	Расширяется до имени файла, указанного в макросе как параметр. Сам макрос записывается в строку как \$NAME(). В скобках вы можете указать файл. Например, макрос \$NAME(\$EDNAME) передаст в запускаемую программу имя файла, расположенного в активном окне Редактора Кода
\$EXT	Развертывается до расширения файла, указанного в макросе как параметр. Сам макрос записывается в строку как \$EXT(). В скобках вы можете указать файл. Например, макрос \$EXT(\$EDNAME) передаст в запускаемую программу имя файла, расположенного в активном окне Редактора Кода
\$EDNAME	Расширяется до полного имени файла, расположенного в активном окне Редактора Кода
\$EXENAME	Расширяется до полного имени выполняемого файла текущего проекта. Например: «C:\PROJ1\UNIT1.EXE». Например: «C:\PROJ1\UNIT1.PAS»
\$PARAMS	Расширяется до командной строки, указанной в диалоге команды Run Parameters
\$PROMPT	Перед запуском программы на выполнение предлагает пользователю диалоговое окно, в котором пользователю предлагается указать значение параметра. Макрос вставляется в строку в виде \$PROMPT(). В скобках вы можете указать пользователю значение параметра по умолчанию, которое и будет показано ему в диалоговом окне при выполнении программы
\$SAVE	Сохраняет на диске файл, активный в Редакторе Кода
\$SAVEALL	Сохраняет текущий проект
\$TDW	Устанавливает окружение для запуска отладчика Turbo Debugger: сохраняет проект, убеждается, что проект скомпилирован с отладочной информацией, повторно компилирует проект, если отладочная информация не была включена в него. Этот макрос имеет смысл использовать, только если вы включили отладчик в меню Tools

Выбрав макрос в списке макросов окна рис. 14.4 б, вы должны нажать кнопку Insert и макрос запишется в позицию курсора в окошке Parameters.

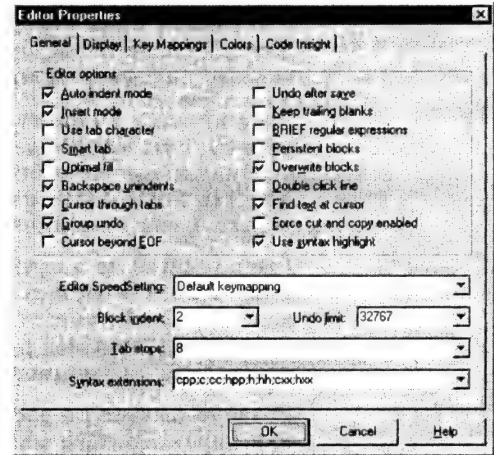
14.2.5 Настройка Редактора Кода

Вызвать настройку Редактора Кода можно щелчком правой кнопки мыши в его окне и выбором команды **Properties** из всплывшего меню. Можно также выполнить для этого команду **Tools | Editor Options**.

В открывшемся окне (рис. 14.5) страницы, относящиеся непосредственно к Редактору Кода: **General**, **Display**, **Key Bindings**, **Color**.

Рис. 14.5.

Страница **General** в окне настройки Редактора Кода



Вид страницы **General** показан на рис. 14.5. Группа **Editor Options** (опции редактора) предлагает ряд опций, которые вы можете включить или выключить используя индикаторы с флажками:

Auto indent mode	При нажатии Enter курсор позиционируется с отступом под первым значащим символом предыдущей не пустой строки
Insert mode	Установка по умолчанию режима вставки, а не замены символа. Эта установка может изменяться пользователем с помощью клавиши Insert
Use tab character	Вставка при переходе к новой строке символа табуляции. Если этот флаг снят, то вставляются пробелы. Если установлен флаг Smart Tab , то данный флаг отключен
Smart tab	Табуляция до первого отличного от пробела символа в предыдущей строке. Если установлен флаг Use Tab Character , то данный флаг отключен
Optimal fill	Оптимальное заполнение минимальным числом символов табуляции и пробелов отступа строки
Backspace unindents	При нажатии клавиши Backspace в начале строки с отступом курсор переходит на отступ предыдущего уровня
Cursor through tabs	Клавиши со стрелками перемещают курсор на следующую позицию табуляции
Group undo	При нажатии клавиш Alt-Backspace или выполнении команды Edit Undo восстанавливается состояние, которое было до последней последовательности команд одного типа

Cursor beyond EOF	Курсор может позиционироваться после символа конца файла
Undo after save	Позволяет восстановить изменения после команды сохранения
Keep trailing blanks	Сохраняет пробелы, набранные вами в конце строки
BRIEF regular expressions	Использование регулярных выражений редактора Brief
Persistent blocks	Сохранение выделения блока даже при сдвиге курсора до тех пор, пока не будет выделен новый блок
Overwrite blocks	Замещение выделенного блока очередным нажатым символом. Если одновременно установлен флаг Persistent Blocks, то вводимый текст добавляется в конец выделенного блока
Double click line	Выделение всей строки при двойном щелчке на каком-нибудь его символе. Если этот флаг не установлен, то при двойном щелчке выделяется слово
Find text at cursor	Текст, на котором стоит курсор, помещается при выполнении команды поиска Search Find в окно задания текста
Force cut and copy enabled	Команды Edit Cut и Edit Copy выполняются, даже если нет выделенного текста
Use syntax highlighting	Выделение цветом синтаксических единиц. Цвета задаются на странице Colors

Все эти опции можно устанавливать независимо друг от друга. Но можно задавать их типичные сочетания, выбирая в выпадающем списке Editor SpeedSetting (быстрая смена стиля редактора) один из пяти предопределенных стилей редактирования: Default Keymapping (стиль по умолчанию), IDE Classic (классическая ИСР), BRIEF emulation (эмуляция редактора BRIEF), Epsilon emulation (эмуляция редактора Epsilon), Visual Studio Emulation (эмуляция редактора Visual Studio). Основные различия между этими пятью стилями определяются установкой следующих опций:

Стиль	Установлены опции
Default Keymapping (стиль по умолчанию)	Auto Indent Mode, Insert Mode, Cursor Through tabs, Group Undo, Overwrite Blocks
IDE Classic (классическая интегрированная среда разработки)	Auto Indent Mode, Insert Mode, Cursor Through Tabs, Group Undo, Persistent Blocks
BRIEF emulation (эмуляция редактора BRIEF)	Auto Indent Mode, Insert Mode, Cursor Through Tabs, Cursor Beyond EOF, Keep Trailing Blanks, Brief Regular Expressions, Force Cut And Copy Enabled
Epsilon emulation (эмуляция редактора Epsilon) и Visual Studio Emulation (эмуляция редактора Visual Studio)	Auto Indent Mode, Insert Mode, Cursor Through Tabs, Group Undo, Overwrite Blocks

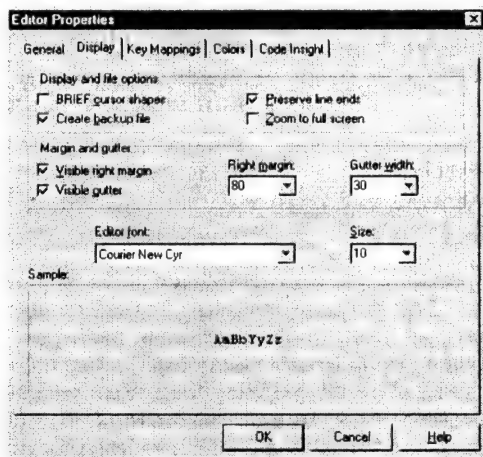
Установки в нижней части диалогового окна позволяют задать следующие значения:

Block indent	Число пробелов в отступах блоков. По умолчанию — 2, максимальное значение — 16
Undo limit	Объем текста, который может быть восстановлен. По умолчанию 32,767 (32K)
Tab stops	Число символов, соответствующих позициям табуляции. Могут перечисляться в возрастающем порядке, с разделителями — запятыми
Syntax extensions	Расширения файлов, в которых используется выделение цветом синтаксических элементов. По умолчанию — .cpp, .c, .cc, .hpp, .h, .hh, .cxx и .hxx

Вид страницы Display окна настройки показан на рис. 14.6. Опции этой страницы определяют отображение текста файлов.

Рис. 14.6.

Страница Display в окне настройки Редактора Кода



Группа Display and file options дает возможность выбрать или отказаться от следующих настроек:

Brief cursor shapes	Установка формы курсора в стиле редактора BRIEF
Create backup file	Сохранять предыдущую — резервную копию файла при сохранении редактируемых файлов. Копии файлов имеют расширения, начинающиеся с символа «~»
Preserve line ends	Сохранять символы конца строки
Zoom to full screen	При разворачивании устанавливать размер окна Редактора Кода во весь экран. Если эта опция выключена, то при разворачивании окно Редактора Кода не заслоняет полосу главного меню и инструментальные панели

Следующий опции определяют шрифт, размер и размещение текста в окне Редактора Кода:

Visible right margin	Делается видимой линия правого поля в окне Редактора Кода
Right margin	Устанавливает позицию правого поля в окне Редактора Кода. По умолчанию длина строки — 80 символов. Эту длину можно увеличивать до 1024
Visible gutter	Делается видимой полоска слева от текста в окне Редактора Кода, в которой вы щелкаете при установке точки прерывания
Gutter width	Устанавливает ширину полоски слева от текста в окне Редактора Кода. По умолчанию — 30
Editor font	Устанавливает шрифт, используемый в Редакторе Кода. Обычно это шрифт постоянной ширины типа Courier. Следите, чтобы этот шрифт имел символы кириллицы
Size	Устанавливает размер шрифта, используемого в Редакторе Кода
Sample	Образец текста выбранного шрифта и размера

Страница Key Mappings (рис. 14.7) позволяет установить комбинации управляющих клавиш Редактора Кода, а также дает возможность сделать доступными или недоступными модули расширений, о которых будет сказано позднее.

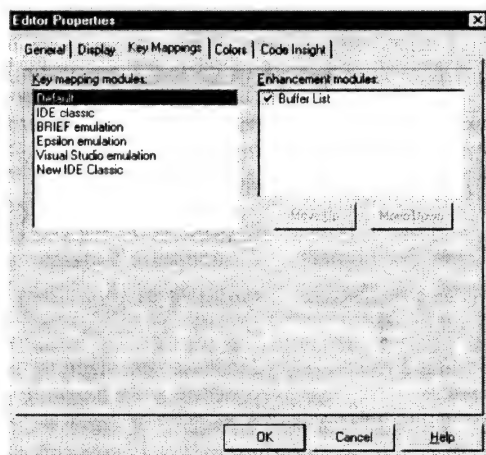
Опция Key mapping modules (управляющие клавиши в стиле редактора ...) имеет шесть предопределенных значений, пять из которых соответствуют тем же стилям, которые устанавливаются на странице General: Default, IDE classic, Brief emulation, Epsilon emulation, Visual Studio emulation, New IDE Classic. Только эти стили в данном случае определяют не поведение редактора, а комбинации управляющих клавиш. Например, комбинация клавиш Ctrl-K-R в стиле по умолчанию или классическом будет считывать блок из файла, в то время как в режиме BRIEF для этой цели используется комбинация Alt-R.

Список Enhancement modules содержит имена установленных модулей расширений. Это специальные пакеты, которые регистрируются в системе и содержат комбинации управляющих клавиш. Вы можете создавать новые модули или изменять содержание уже установленных, используя API Open Tools. Индикаторы возле соответствующих модулей в списке позволяют делать модули доступными или недоступными.

На странице Color (цвета) вы можете определить вид выделения различных синтаксических элементов текста вашей программы. Как и на двух предыдущих

Рис. 14.7.

Страница Key Mappings в окне настройки Редактора Кода



страницах диалогового окна, здесь имеется возможность быстрого выбора настройки из списка Color Speed Setting, где вам предоставляется выбор из predetermined-ных цветовых схем. Вы можете также самостоятельно выбрать установки цветов отдельных элементов, указывая их в списке Element. Результат всех ваших изменений в установках можно видеть в имитации окна редактора в нижней части диалогового окна.

Можно упомянуть еще об одной настройке, введенной в C++Builder 5. Вы можете задать формат разделительных линий, возникающих в коде, загруженном в окно Редактора Кода и разделяющем друг от друга отдельные функции. Чтобы задать свой формат, надо внести в текстовый файл **bcb.bcf**, размещенный в каталоге **Bin**, раздел [Code Formatting] и после этого заголовка поместить желательный вид разделительной линии. Например:

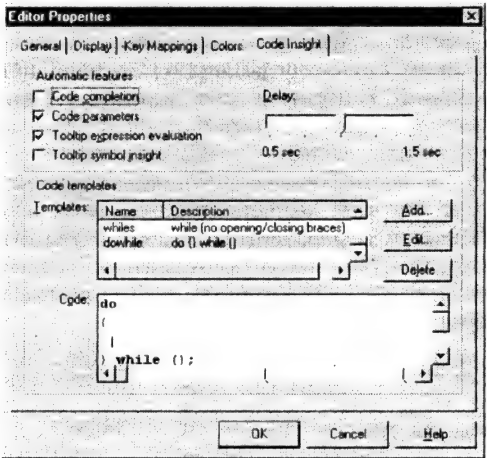
```
[Code Formatting]
Divider Line=// ***** Разделитель *****
```

Только учтите, что ваше изменение скажется только начиная с очередной загрузки C++Builder.

14.2.6 Настройка Code Insight — Знатока Кода

Для настройки Code Insight — средства, обеспечивающего подсказки и помощь при написании и отладке кодов (см. раздел 2.5.3.1), надо выполнить команду Tools | Editor Options и затем перейти на страницу Code Insight (рис. 14.8).

Рис. 14.8.
Страница Code Insight в окне настройки Редактора Кода



В верхней части окна расположены индикаторы опций автоматического выполнения различных функций Code Insight:

Code Completion	Завершение кода — подсказка в виде списка свойств, методов, событий, относящихся к данному компоненту
Code Parameters	Подсказка параметров функций, процедур, методов
Tooltip Expression Evaluation	Оценка выражений во время останова или пошагового выполнения приложения во время его отладки
Tooltip Symbol Insight	Подсказка определений идентификаторов, над которыми перемещается курсор мыши

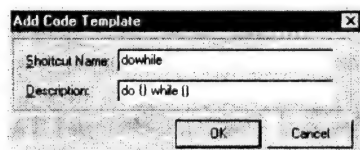
По умолчанию автоматическое выполнение всех этих опции включено. Но вы можете установить в автоматический режим только те, которые вам постоянно нужны. Например, после того, как вы несколько освоились с C++Builder, можно отключить автоматическое выполнение опции Code Completion. Это исключит автоматическое появление подсказок свойств, методов и событий компонентов, поскольку эти подсказки в некоторых случаях мешают нормальной записи кода, а иногда приводят и к невольным ошибкам, подставляя в ваш код неправильные свойства или методы. При отключении автоматического выполнения этой опции вы все равно можете воспользоваться ею в любой момент, нажав клавиши Ctrl-пробел. Может оказаться также полезным отключить автоматическое выполнение опции ToolTip Symbol Insight, поскольку она занимает заметное время во время работы с кодом и замедляет реакцию C++Builder на ваши действия. В тех сравнительно редких случаях, когда вам понадобится эта опция, вы можете опять подключить ее.

Ползунок Delay устанавливает задержку автоматического срабатывания Code Insight. Это то время, которое отводится вам для самостоятельного продолжения кода. Если вы задержались на большее время, появится подсказка.

Раздел Code Templates содержит список шаблонов типичных структур языка C++, подсказки по которым предлагает Code Insight. Если вы выделите название и краткое описание одного из шаблонов в списке, то в нижнем окне сможете увидеть предлагаемый шаблон. Кнопка Edit позволяет вам изменить имя и краткое описание шаблона. Сам текст шаблона вы можете изменять, не нажимая этой кнопки, а просто переведя курсор в нижнее окно с кодом шаблона. Кнопка Delete позволяет удалить шаблон из списка. Кнопка Add дает вам возможность добавить в список новый шаблон. Пусть, например, вы хотите добавить шаблон управляющей структуры **do...while** (такой шаблон в C++Builder не встроен). Нажмите кнопку Add, и вам будет показано окно (рис. 14.9), в котором вы можете ввести имя шаблона Shortcut Name (на рисунке — **dowhile**) и его краткое описание Description. Затем вы можете ввести текст шаблона (этот текст вы можете видеть на рис. 14.8).

Рис. 14.9.

Окно добавления и редактирования шаблона



В тексте шаблона вы можете вставить вертикальную черту в том месте, в котором остановится курсор при вводе этого шаблона в текст. Наверное, эту черту следует поставить в том месте, где пользователь должен будет внести первый элемент, заполняющий шаблон. Сама черта в этом шаблоне видна не будет.

14.2.7 Настройка Исследователя Классов ClassExplorer

Исследователь Классов ClassExplorer показывает дерево всех типов, классов, свойств, методов, глобальных переменных и глобальных функций, содержащихся в модуле, открытом в Редакторе Кода. В C++Builder 5 этот инструмент облегчает также внесение в классы новых методов, свойств, сообщений.

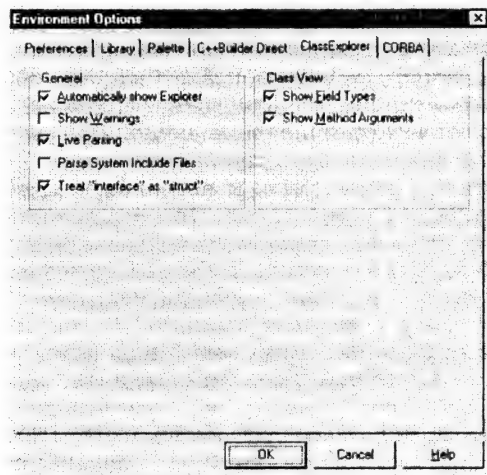
Настройка Исследователя Классов производится командой Tools | Environment Options и переходом на страницу ClassExplorer (рис. 14.10).

Опция Automatically show Explorer, определяет автоматическое появление окна Исследователя Классов, встроенного в окно Редактора Кода. По умолчанию эта опция включена, но, вероятно, во многих случаях ее лучше отключить, чтобы не уменьшать площадь видимого окна кода. Вы всегда при необходимости можете и в этом случае вызвать Исследователя Кода командой View | ClassExplorer.

Опция Show Warnings определяет отображение в ClassExplorer сообщений об ошибках, вследствие которых исходный код невозможно прочитать.

Рис. 14.10.

Страница ClassExplorer окна
Environment Options



Опция Live Parsing обеспечивает обновление информации в окне Исследователя Кода при модификации проекта в окне Редактора Кода. Если эта опция выключена, то обновление информации происходит только когда проект сохраняется.

Опция Parse System Include Files обеспечивает обработку всех файлов, которые указаны в проекте директивами `#include`. Если эта опция выключена (а она выключена по умолчанию), то обрабатываются только файлы, указанные директивой `#include <file.h>`, а системные файлы, указанные директивами `#include <file.h>`, не обрабатываются. Включение опции может заметно замедлить работу, поскольку в проект может включаться очень много системных файлов.

Опция Treat «interface» as «struct» обеспечивает интерпретацию интерфейса как структуры.

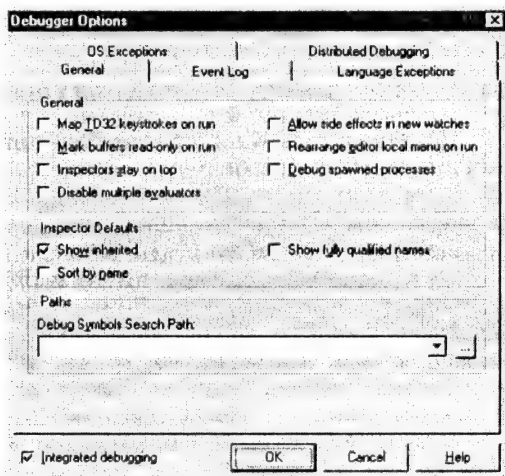
Опция Show Field Types задает отображение в дереве класса типов полей. А опция Show Method Arguments отображает в дереве аргументы всех методов.

14.2.8 Настройка отладчика

Вызов настройки отладчика осуществляется командой Tools | Debugger Options. Появляется многостраничное окно настройки, показанное на рис. 14.11.

Рис. 14.11.

Страница General окна настройки
отладчика



Основная опция этого окна расположена внизу и видна на любой странице. Это опция *Integrated debugging*, обеспечивающая активацию отладчика. Если выключить эту опцию, то ни одна из возможностей отладчика не будет реализовываться.

На странице *General*, показанной на рис. 14.11, имеются следующие опции, определяющие интерфейс пользователя во время работы отладчика:

Map TD32 keystrokes on run	Позволяет использовать клавиши карты TD32 в процессе выполнения приложения. Установка этой опции автоматически устанавливает и делает недоступной опцию <i>Mark buffers read-only on run</i>
Mark buffers read-only on run	Все редактируемые файлы помечаются во время выполнения приложения как файлы только для чтения. Это не позволяет вам во время выполнения, перейдя в ИСР, что-то изменить (или случайно испортить) в файлах. После завершения приложения атрибуты файлов восстанавливаются
Inspectors stay on top	Оставляет все окна отладчика видимыми, даже если они не активны
Disable multiple evaluators	Запрещает работу сразу двух блоков оценки C++ и Pascal. Если эта опция установлена, то работает только блок оценки C++
Allow side effects in new watches	Разрешает побочные эффекты (например, изменение переменных) в процессе наблюдения (см. раздел 2.6.4)
Rearrange editor local menu on run	Перестраивает контекстное меню Редактора Кода во время выполнения приложения, вынося на верх разделы, связанные с отладчиком. Это ускоряет доступ к командам отладки
Debug spawned processes	Автоматически отлаживает процесс, порожденный отлаживаемым. Если эта опция не установлена, порожденный процесс запускается, но не отлаживается

Опции группы *InspectorDefaults* определяют опции по умолчанию окна Инспектора Отладки (см. раздел 2.6.8). Опция *Show inherited* обеспечивает отображение на страницах окна всех свойств и методов, как объявленных в данном классе, так и наследуемых. Если эта опция выключена, то отображается только то, что объявлено в данном классе. Опция *Sort by name* упорядочивает отображаемые данные в алфавитной последовательности имен. Опция *Show fully qualified names* обеспечивает отображение наследуемых элементов с их полными именами.

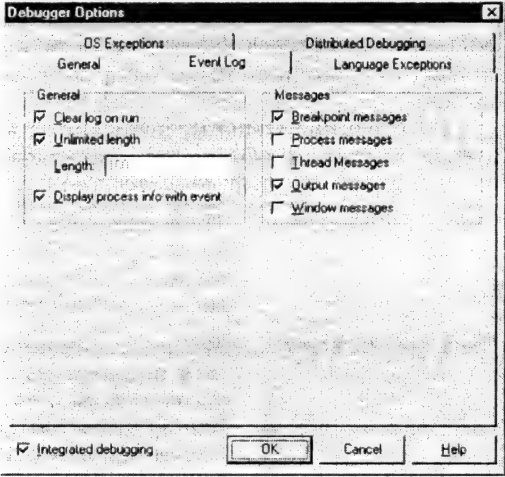
Все эти опции могут быть изменены при работе с Инспектором Отладки (см. раздел 2.6.8).

Окно *Debug Symbols Search Path* определяет каталог, в котором сохраняются файлы символов, используемых при отладке: *.tds*, *.rsm*, *.dcp*. Если окно оставлено незаполненным, то эти файлы хранятся вместе с выполняемым файлом проекта *.exe*.

Страница *Event Log* окна настройки отладчика (рис. 14.12) позволяет установить опции сообщений о событиях. Протокол этих событий, сопровождающих выполнение вашего приложения, вы можете посмотреть в процессе выполнения или после его окончания, выполнив команду *View | Debug Windows | Event Log* или нажав клавиши *Ctrl-Alt-E* (см. раздел 2.6.9).

Следующие опции страницы *Event Log*, объединенные в разделе *General*, определяют способ отображения событий:

Рис. 14.12.
Страница Event Log окна настройки отладчика



Clear log on run	Очистка протокола при начале очередного сеанса отладки
Unlimited Length	Неограниченная длина отображения событий
Length	Максимальная длина отображения сообщения о событии. Эта опция недоступна, если выбрана опция Unlimited

Опции раздела Messages определяют типы сообщений, которые будут заноситься в протокол событий:

Breakpoint messages	Сообщения о прерываниях выполнения из-за вставленных вами в проект точек прерывания или из-за генерации исключений
Process messages	Сообщения о загрузке и окончании всех процессов
Thread Messages	Сообщения нитей (параллельных потоков) приложений
Output messages	Сообщения, генерируемые функцией OutputDebugString (см. раздел 2.6.9)
Window messages	Сообщения Windows

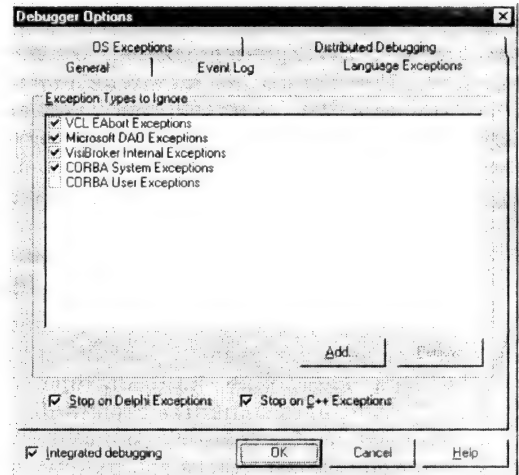
Для пользователей, не собирающихся погружаться в тонкости работы системы, можно рекомендовать установить только опции Breakpoint messages и Output messages. Это позволит вам отследить последовательность выполнения вашего приложения и с помощью функции **OutputDebugString** вывести сообщения, детализирующие состояние приложения в различные моменты времени.

Страница Language Exceptions (рис. 14.13) позволяет управлять прерываниями отладки при генерации исключений в приложении. Выключение опций Stop on Delphi Exceptions и Stop On C++ Exceptions (останов при генерации исключений Delphi или C++) обеспечивает отсутствие остановов при отладке, сопряженных с появлением дополнительного окна сообщения об исключении. Иначе говоря, приложение в процессе отладки будет вести себя так же, как оно ведет себя при обычном запуске.

Если опции Stop on Delphi Exceptions и Stop On C++ Exceptions включены, то остановы отладчика будут при любых исключениях, кроме тех, которые перечислены в окне Exception Types to Ignore, или их потомков. По умолчанию в список исключений, при которых не происходит остановка отладчика, входят те, которые вы види-

Рис. 14.13.

Страница Language Exceptions окна
настройки отладчика



те в списке на рис. 14.13, да и то не все из них отмечены. С помощью кнопки Add вы можете внести в список другие исключения и пометить их. Например, если вы внесете в список исключение **EMathError**, то приложение в процессе отладки не будет останавливаться при делении на нуль, переполнении и т.д.

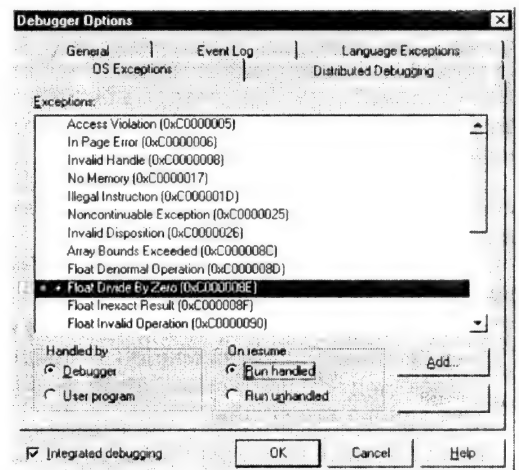
Кнопка Remove позволяет удалить из списка выделенное в нем исключение.

Страница OS Exceptions (рис. 14.14) определяет опции обработки исключений. Вверху страницы расположен список исключений. Вы можете добавить в него свои исключения, пользуясь кнопкой Add. Кнопка Remove позволяет удалить выделенное исключение из списка, но удалять можно только исключения, определенные пользователем. Для предопределенных в C++Builder исключений кнопка Remove недоступна.

Радиокнопки Handled by определяют, чем будет обрабатываться исключение: Debugger — отладчиком, User program — программой пользователя. Радиокнопки On resume определяют, будет ли C++Builder продолжать обработку исключения (Run handled), или нет (Run unhandled). Выбор кнопки Debugger приводит к появлению около имени данного исключения красного кружочка, а выбор кнопки Run Handled — к появлению зеленой стрелочки. На рис. 14.14 вы можете это видеть около исключения Float Divide By Zero. Сочетание этих кнопок приводит к тому, что стандартное сообщение об исключении в процессе отладки не появляется, даже

Рис. 14.14.

Страница OS Exceptions окна настройки
отладчика



если вы, вопреки предположению не предусмотрели собственную обработку. Впрочем, учтите, что все это относится только к отладке. Если вы выполните приложение не из C++Builder, то на его работу все это никак не повлияет.

Страница Distributed Debugging связана с отладкой распределенных приложений, которые мы в данной книге не рассматриваем.

14.2.9 Настройка компилятора и компоновщика

14.2.9.1 Рекомендации по сокращению времени компиляции, компоновки и загрузки приложения

C++Builder предлагает вам ряд опций, обеспечивающих сокращение временных затрат на цикл редактирования и выполнения приложения. Они могут задаваться при настройке компилятора и компоновщика с помощью команды Project | Options на различных страницах открывающегося диалогового окна.

Для того, чтобы грамотно пользоваться этими опциями, полезно представлять себе, что происходит, когда вы, например, выполняете команду Project | Run или нажимаете клавишу F9. При этом C++Builder выполняет следующие операции:

1. Просмотр файлов вашего проекта с целью определить, какие из них нуждаются в обработке.
2. Компиляция всех файлов **.cpp**, **.pas**, **.asm**, **.rc**, которые были изменены со времени последней компиляции.
3. Компоновка всех объектных файлов в выполняемый файл.
4. Загрузка выполняемого файла в память и его выполнение.

Для ускорения этих операций C++Builder использует несколько возможностей. Прежде всего это выборочная компиляция файлов. При первой в данном сеансе компиляции и компоновке приложения C++Builder запоминает для каждого файла отметку времени его изменения. Эта информация используется при очередной компиляции для определения того, какие объектные файлы нуждаются в перестроении.

Компилируемая программа может указать, что некоторое множество заголовочных файлов, включенных в проект директивами **#include**, должно быть предварительно скомпилировано и сохранено в отдельном файле. Этот файл загружается при очередной компиляции и таким образом удается избежать повторной компиляции этих заголовочных файлов. Более того, C++Builder кэширует этот файл в памяти компьютера, так что даже повторная загрузка его может не требоваться.

Пошаговый интеллектуальный компоновщик ILINK32 сохраняет информацию о произведенной компоновке в файлах выборочной компоновки **.il?** и при последующей компоновке загружает эту информацию. В результате заново компонуются только те файлы, которые были изменены. При этом компонуются только те функции и переменные из подключаемых вами библиотек, которые действительно используются в проекте.

Выполняемый файл **.exe** формируется непосредственно в памяти. Это исключает затраты времени на его сохранение на диске и последующую загрузку в память для выполнения.

Ниже приведен перечень мер, которые вы можете принять для сокращения временных затрат на весь цикл редактирования и выполнения приложения.

1. Не используйте без особой необходимости команду Build All. Эта команда удаляет из памяти информацию о предыдущей компоновке, удаляет результаты предварительной компиляции заголовочных файлов, компилирует и компонуют заново все файлы.

2. Сохранение информации о последней произведенной компоновке существенно сокращает время компиляции и компоновки приложения, но может требовать немалого пространства на диске. Создание этих файлов выборочной компоновки `.il?` можно отключить, выключив опцию `Don't Generate State Files` на странице `Linker`. При работе с одним проектом вся информация хранится в памяти. Поэтому включение или выключение этой опции не влияет на время компоновки. Но если вы работаете с группой проектов, поочередно компилируя их, то хранение в файле информации о предыдущей компоновке существенно сокращает время последующей компоновки. На время первой компоновки каждого из проектов опция `Don't Generate State Files` не влияет.
3. При работе в `Windows NT` на компьютерах с оперативной памятью 32 мегабайта и менее ускорить запуск приложения после его компоновки помогает включение опции `In-memory .EXE` на странице `Linker`, обеспечивающей формирование выполняемого модуля в памяти без записи на диск и, соответственно, без необходимости его последующей загрузки с диска. `Windows 95` эту опцию не поддерживает.
4. Для больших проектов существенное затягивание процессов компиляции и компоновки может вызвать информация отладчика `C++Builder`. Если вы не планируете при очередных прогонах приложения использовать информацию отладчика, можно отключить ее кнопкой `Release` на странице `Compiler` (рис. 14.15). В этих случаях имеет также смысл отключить интегральный отладчик — снять флажок индикатора `Integrated debugger` в окне `Debugger Options`, открываемом командой `Tools | Debugger Options`. Все это не только сократит время компиляции, компоновки, загрузки и выполнения, но и уменьшит затраты пространства на диске.
5. В `C++Builder 5` введена возможность установки в Менеджере Проектов опций компиляции локально для отдельных модулей (см. раздел 2.4.3). Это можно использовать в больших проектах, отключая информацию отладчика в уже отлаженных модулях и оставляя ее в отлаживаемых в данный момент. Это, в частности, позволит существенно уменьшить затраты дискового пространства.
6. Существенного сокращения времени компиляции и уменьшения затрат памяти можно добиться, используя версию библиотеки компонентов `VCL`, не предусматривающую отладку. Для этого надо на странице `Linker` не включать опцию `Use debug libraries`.
7. Заметное сокращение времени компиляции а также упрощение отладки получается, если на странице `Compiler` выключить опции `Code optimization` — оптимизацию приложения по быстрдействию.
8. Кардинальный способ сократить затраты времени при работе с большими проектами — увеличить оперативную память вашего компьютера до 48 – 64 мегабайт.

14.2.9.2 Быстрая настройка компилятора и компоновщика

Настройка компилятора и компоновщика осуществляется с помощью команды `Project | Options` на различных страницах открывающегося диалогового окна `Project Options: Compiler, Advanced Compiler, Pascal, Linker, Tasm`. Если вы не хотите вдаваться в особенности применения различных опций, то наиболее простой способ настройки — использование кнопок `Full debug` и `Release` на странице `Compiler` (рис. 14.15). Эти кнопки служат для быстрой установки совокупности опций страниц `Compiler, Advanced Compiler, Pascal, Linker, Tasm`. Совокупность опций, соответствующую кнопке `Full debug`, можно рекомендовать в процессе отладки приложения, когда не надо особо заботиться о скорости выполнения, но требуется включение всех возможностей отладки. Правда, при этом могут возникнуть некоторые

сложности, связанные с длительностью процессов компиляции и отладки и с затратами дисковой памяти, о которых говорилось в разделе 14.2.9.1. Кнопку Release целесообразно использовать после того, как проект отлажен и готов к эксплуатации. Она позволяет оптимизировать выполняемый файл.

Ниже даются списки опций, устанавливаемых кнопками Full debug и Release. Пояснения этих опций вы найдете в следующих разделах.

Кнопка Full debug устанавливает опции:

Страница	Опции
Compiler	Enables Code optimizations None Enables Debugging Debug information Enables Debugging Line number information Enables Debugging Disable inline expansions Enables Compiling Stack frames
Advanced Compiler	Enables Register Values None
Pascal	Disables Code generation Optimization Enables Code generation Stack frames Enables Debugging Debug information Enables Debugging Local symbols Enables Debugging Symbol info
Linker	Enables Linking Include debug information
Tasm	Enables Debug Information Full

Опции, устанавливаемые кнопкой Release:

Страница	Опции
Compiler	Enables Code optimizations Speed with scheduling Disables Debugging Debug information Disables Debugging Line number information Disables Debugging Disable inline expansions Disables Compiling Stack frames
Advanced Compiler	Enables Register Variables Automatic
Pascal	Enables Code generation Optimization Disables Code generation Stack frames Disables Debugging Debug information Disables Debugging Local symbols Disables Debugging Symbol info
Linker	Disables Linking Include debug information
Tasm	Enables Debug Information None

В следующих разделах страницы настройки компилятора и компоновщика будут рассмотрены более подробно. Но прежде, чем рассматривать отдельные страницы, следует указать, что при работе с любой страницей в левом нижнем углу окна

виден индикатор Default (см., например, рис. 14.15). Если его включить, то все сделанные установки будут сохраняться как установки по умолчанию для всех последующих проектов.

14.2.9.3 Страница Compiler

Эта страница, устанавливающая основные опции настройки компилятора C++, показана на рис. 14.15. Опции на ней скомпонованы в несколько групп.

Назначение основных кнопок страницы — Full debug и Release в общих чертах было рассмотрено в предыдущем разделе.

Группа опций Code Optimization на странице Compiler включает и выключает оптимизацию кода с точки зрения скорости выполнения. Опция None, которая включена по умолчанию, выключает любую оптимизацию. Это способствует ускорению компиляции и не создает сложностей в отладке, связанных с тем, что какие-то переменные и операторы кода могут быть удалены из него в результате оптимизации.

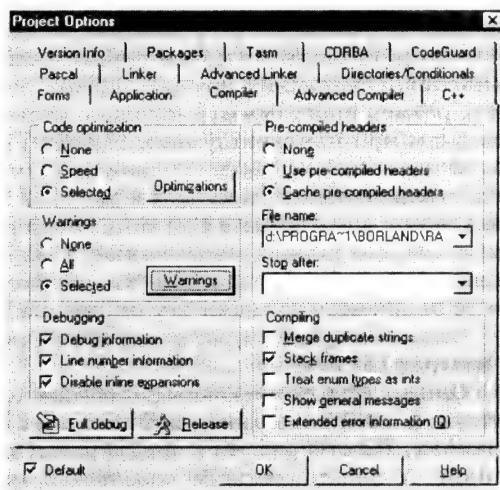
Радиокнопка Speed устанавливает следующую группу опций оптимизации:

- Inline intrinsic functions (встраивание функций **inline**) — эквивалент опции командной строки **-Oi**. В результате ряд небольших функций встраивается в код, сокращая затраты времени на их вызов. Это ведет к более быстрому выполнению приложения, но увеличивает размер выполняемого файла.
- Induction variables (индуцирование переменных) — эквивалент опции командной строки **-Ov**. В результате оптимизируются фрагменты кода, связанные с циклами.
- Optimize common subexpressions (оптимизация общих фрагментов выражений) — эквивалент опции командной строки **-Og**. Эта опция удаляет из кода каждой функции повторяющиеся в ней выражения, сохраняя их значения и исключая таким образом повторяющиеся вычисления. Это сокращает время вычислений, но редко сокращает объем выполняемого файла.

Радиокнопка Selected и нажатие кнопки Optimization вызывают диалоговое окно оптимизации, в котором вы можете отдельно задать или отключить перечисленные выше опции. Кроме того в этом окне вы можете установить опцию Pentium scheduling — эквивалент опции командной строки **-OS**, позволяющую учесть при оптимизации особенности компьютеров Pentium. Кнопка Defaults в этом окне восстанавливает состояние всех опций оптимизации по умолчанию.

Рис. 14.15.

Страница Compiler окна опций проекта



Группа опций Warnings задает типы предупреждений компилятора, которые будут им выдаваться. Опция All (установлена по умолчанию) обеспечивает выдачу всех возникающих при компиляции предупреждений. Опция None запрещает выдачу любых предупреждений. Опция Selected и нажатие кнопки Warnings вызывают диалоговое окно, в котором вам предлагается длинный список типов предупреждений, любой из которых вы можете включить или выключить.

Группа опций Debugging задает генерацию различного вида отладочной информации в процессе компиляции. Опция Debug information добавляет в объектные файлы .obj отладочную информацию. Опция Line number добавляет в объектные файлы .obj информацию о номерах строк исходного файла. Опция Disable inline expansions запрещает компилятору разворачивать встраиваемые inline функции. При этом встраиваемые функции генерируются и вызываются как и все иные. Это облегчает отладку приложения.

По умолчанию все опции группы Debugging включены. Впрочем, они будут обеспечивать встраивание отладки в выполняемый модуль только в случае, если одновременно на странице linker включена опция Include Debug Information. Если ваши объектные файлы .obj получаются очень большими, вы можете уменьшить их размер, отключая те или иные опции отладки. При завершении работы с проектом эти опции безусловно надо выключить и оттранслировать проект заново.

Опции группы Pre-compiled headers определяют предварительную компиляцию заголовочных файлов. Эта возможность компилятора, рассмотренная в разделе 14.2.9.1, может существенно ускорять повторную компиляцию приложения за счет того, что сохраняет в файле на диске таблицу символов использованных заголовочных файлов. Но это может быть связано с большими объемами файлов на диске. Опция None запрещает генерацию файла предварительной компиляции заголовочных файлов. Опция Use pre-compiled headers обеспечивает генерацию и использование этого файла (его имя по умолчанию — ...\\lib\\vcl.csm). Опция Cache pre-compiled headers (включена по умолчанию) обеспечивает кэширование файла предварительной компиляции. Это полезно, если предварительно компилируется более одного заголовочного файла. Окно File Name позволяет вам изменить имя по умолчанию файла предварительной компиляции. Это имя можно задать или с полным путем, или с путем относительно каталога C++Builder, обозначаемого макросом \$(BCB). Окно Stop After прерывает создание файла предварительной компиляции после того, как скомпилирован указанный в этом окне файл. Это можно использовать, чтобы сократить размер файла предварительной компиляции.

Группа индикаторов Compiling устанавливает основные опции компилятора, которые отражаются на компиляции большинства приложений C++Builder. Опция Merge duplicate strings дает компилятору возможность объединять две строки символов, следующих друг за другом, в одну строку. Это может несколько сокращать размер программы, несколько удлинять ее компиляцию, а иногда приводит к ошибкам при изменении вами одной из строк.

Опция Stack frames (включена по умолчанию) заставляет компилятор генерировать стандартный стек входов и выходов функций, что облегчает отладку по шагам с заходом в вызываемые функции. Если выключить эту опцию, то все функции, не имеющие локальных переменных и параметров, компилируются с укороченной адресацией входов и выходов. Это сокращает код и ускоряет выполнение. Поэтому после окончания отладки эту опцию следует выключать.

Опция Treat enum types as ints обеспечивает размещение переменных типа enum в четырех байтах.

Опция Show general messages обеспечивает отображение всех сообщений компилятора и компоновщика, помимо предупреждений, замечаний и сообщений об ошибках. На отображение предупреждений и сообщений об ошибках эта опция не влияет. А дополнительные сообщения вряд-ли дадут вам что-нибудь ценное.

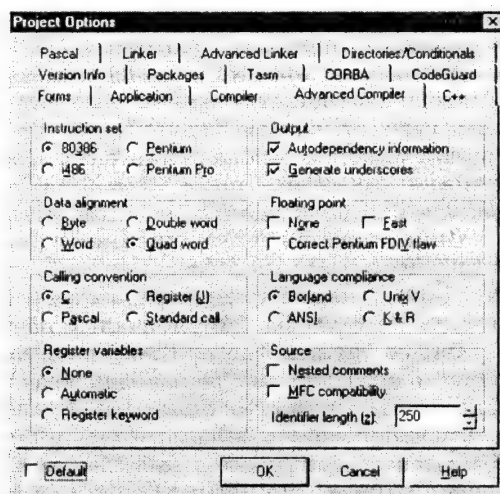
Опция Extended error information обеспечивает расширенную информацию о контексте ошибок. Около сообщений появляются индикаторы с символом «+», которые можно раскрыть для получения дополнительной информации.

14.2.9.4 Страница Advanced Compiler

Эта страница устанавливает дополнительные опции компилятора C++, необходимые для приложений, использующих библиотеки компонентов (VCL). Вы можете изменять установки на этой странице, но при этом надо соблюдать определенную осторожность, поскольку необходимо обеспечивать совместимость с VCL. Общий вид страницы показан на рис. 14.16.

Рис. 14.16.

Страница Advanced Compiler
окна опций проекта



Опции на странице скомпонованы в несколько групп. Опции группы Instruction Set определяют тип компьютера, для которого компилируется приложение: 80386 (по умолчанию), i486, Pentium, Pentium Pro.

Опции группы Data Alignment определяют выравнивание данных в памяти: Byte — выравнивание по границам 8 бит, Word — 16 бит, Double Word — 32 бита, Quad Word — 64 бита.

Опции группы Calling Convention определяют соглашение, используемое при вызове функций. Соглашения различаются способами обработки стеков, последовательностью параметров, чувствительностью к регистру, префиксами глобальных переменных.

Опция C (включена по умолчанию) соответствует соглашениям языка C — в частности, определяет чувствительность к регистру. Включение этой опции эквивалентно объявлению всех функций с ключевым словом `__cdecl`. Функции, соответствующие соглашению C, могут воспринимать список параметров переменной длины. Если вы установили эту опцию, то функции, использующие другие соглашения, можете объявлять ключевыми словами `__pascal`, `__fastcall`, `__stdcall`.

Опция Pascal соответствует соглашениям языка Pascal — в частности, определяет перевод идентификаторов к верхнему регистру. Включение этой опции эквивалентно объявлению всех функций с ключевым словом `__pascal`. В результате вызов функций обычно короче и производится быстрее, чем при опции C. В функции должен передаваться список параметров указанной длины и типов. Если вы установили эту опцию, то функции, использующие другие соглашения, можете объявлять ключевыми словами `__cdecl`, `__fastcall`, `__stdcall`.

Опция Register (J) соответствует соглашениям передачи параметров Register. Включение этой опции эквивалентно объявлению всех функций с ключевым словом **__fastcall**. В результате, где возможно, параметры будут передаваться в регистры. Если вы установили эту опцию, то функции, использующие другие соглашения, можете объявлять ключевыми словами **__cdecl**, **__pascal**, **__stdcall**.

Опция Standard Call соответствует соглашениям передачи параметров Stdcall. Включение этой опции эквивалентно объявлению всех функций с ключевым словом **__stdcall**. В функции должен передаваться список параметров указанной длины и типов. Если вы установили эту опцию, то функции, использующие другие соглашения, можете объявлять ключевыми словами **__cdecl**, **__pascal**, **__fastcall**.

Группа опций Register Variables определяет, могут ли локальные переменные во время выполнения храниться в системных регистрах. Опция None (включена по умолчанию) запрещает хранение локальных переменных в регистрах, даже если вы используете для них ключевое слово **register**. Опция Automatic разрешает компилятору перемещать переменные в регистры, даже если вы не указали ключевого слова **register**. Опция Register Keyword разрешает использовать регистр для переменной, только если она объявлена с ключевым словом **register** и ее размещение в регистре возможно. Опции Automatic и Register Keyword обеспечивают более быстрое выполнение, оптимизируя использование регистров, но могут затруднять отладку.

Опции группы Output определяют, надо ли включать в информационный файл Makefile информацию о генерируемых объектных модулях (опция Autodependency Information) и надо ли предварять имена функций символом подчеркивания (опция Generate Underscores). По умолчанию обе опции включены.

Опции группы Floating Point позволяют оптимизировать производительность и точность вычислений с плавающей запятой. По умолчанию все опции этой группы выключены. Опция None запрещает использование плавающей запятой. При этом нельзя использовать библиотеки функций с плавающей запятой. Если вы все-таки обратитесь в программе к вычислениям с плавающей запятой, вам будет выдана ошибка компоновки. Опция Fast разрешает оптимизировать вычисления с плавающей запятой независимо от явного или неявного задания преобразований типов. Если эта опция выключена, то компилятор придерживается стандарта ANSI для преобразования типов. Включение этой опции может ускорить вычисления. Опция Correct Pentium FDIV flow позволяет исправить ошибки деления с плавающей запятой, которые были свойственны ранним кристаллам Pentium.

Опции группы Language Compliance определяют множество воспринимаемых компилятором ключевых слов. Ключевые слова, не соответствующие установленному множеству, воспринимаются компилятором как обычные идентификаторы.

По умолчанию включена опция Borland, соответствующая расширению ключевых слов, принятому в корпорации Inprise. Это расширение включает, в частности, ключевые слова **near**, **far**, **huge**, **asm**, **cdecl**, **pascal**, **interrupt**, **__export**, **__ds**, **__cs**, **__ss**, **__es**, псевдопеременные регистров (**_AX**, **_BX** и т.д.). Если в вашем коде обнаружены синтаксические ошибки, связанные с непониманием компилятором ключевых слов, проверьте прежде всего, включена ли опция Borland.

Опция ANSI соответствует стандарту языков C и C++. Поэтому данная опция обеспечивает максимальную переносимость вашего кода на другие платформы. Опция Unix V устанавливает распознавание только ключевых слов UNIX V. Опция K & R устанавливает распознавание только ключевых слов расширения K&R.

Группа опций Source управляет интерпретацией кода. Опция Nested Comments разрешает наличие в коде C и C++ вложенных комментариев. В стандарте C вложенные комментарии не допускаются, так что их использование делает код непореносимым на другие платформы. Поэтому данная опция по умолчанию выключена.

Опция MFC Compatibility позволяет компилировать код так, что он делается совместимым с классами Microsoft — Microsoft foundation classes (MFC).

Окно Identifier Length (z) позволяет указать число символов, которые компилятор будет распознавать в идентификаторах. В отличие от классического C++, в котором распознаются идентификаторы неограниченной длины, C++Builder распознает не более заданного числа символов идентификаторов, включая идентификаторы переменных, имена макросов препроцессора, имена элементов структур. Длину идентификаторов можно указать от 8 до 250. Если задать значение 0, оно будет трактоваться как 250. Учтите, что другие системы, в частности, компилятор UNIX, учитывает только 8 первых символов идентификатора. Так что если вы хотите написать код, переносимый на подобную платформу, лучше задать в окне Identifier Length (z) соответствующее число символов. Это поможет при переносе на другую платформу избежать ошибок, связанных с усечением длинных идентификаторов, когда разные идентификаторы начнут трактоваться как идентичные.

14.2.9.5 Страница C++

Страница C++ устанавливает опции компилятора, специфические для C++ и необходимые для приложений, использующих библиотеку компонентов VCL. Изменять настройки этой страницы можно только для приложений, не использующих VCL. Если же вы используете VCL, то изменять эти настройки нельзя. Поэтому ограничимся только кратким описанием отдельных групп опций.

Опции группы Member Pointers определяют виды указателей на размещенные в памяти элементы классов. Установленная по умолчанию опция All cases разрешает доступ к любым элементам без ограничения, что в некоторых частных случаях может быть не самым эффективным.

Опции группы Compatibility определяют обратную совместимость с прежними версиями стандарта ANSI. Они разрешают вам перекомпилировать прежние библиотеки. По умолчанию эти опции выключены.

Опции группы Virtual Tables позволяют оптимизировать размер таблиц виртуальных функций. Опция Templates определяет обработку компилятором шаблонов. Опции группы Exception Handling управляют обработкой прерываний и информации RTTI. Опция General устанавливается для совместимости с объектами библиотек, реализующими пустые базовые классы нулевого размера.

В большинстве случаев все опции этой страницы лучше не трогать, пока у вас не возникло каких-то проблем и нет уверенности, что опции этой страницы могут их решить.

14.2.9.6 Страница Pascal

Эта страница устанавливает опции компилятора Object Pascal. Они влияют на компиляцию модулей на языке Object Pascal, например, модулей, заимствованных из Delphi 5. Общий вид страницы показан на рис. 14.17.

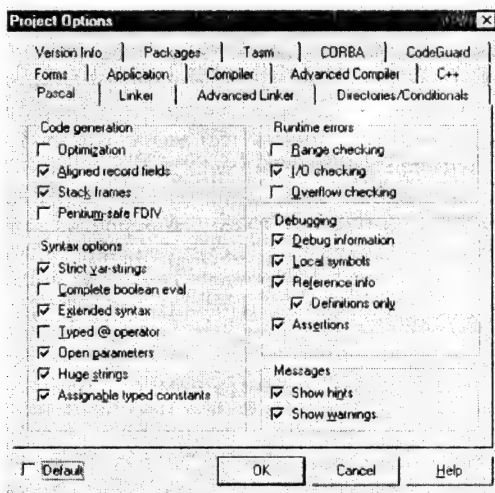
Опции данной страницы соответствуют ключевым директивам компилятора языка Object Pascal. Установка той или иной опции эквивалентна включению соответствующей директивы. Подробнее об Object Pascal в Delphi 5 и о самой системе Delphi 5 вы можете узнать в книге А.Я. Архангельского «Программирование в Delphi 5», издательство БИНОМ, 2000, или в серии книг «Все о Delphi».

Опции на странице Pascal скомпонованы в несколько групп. Опции группы Code generation влияют на способ компиляции. Опция Optimization (эквивалент директивы `{SO}`) разрешает оптимизацию компиляции. Опция Aligned record fields (эквивалент директивы `{SA}`) выравнивает по границам 32 байтов поля записей. Опция Stack frames (эквивалент директивы `{SW}`) генерирует стек для всех процедур и функций. Опция Pentium-safe FDIV (эквивалент директивы `{SU}`) генерирует коды вычислений с плавающей запятой, работающие на всех процессорах Pentium, включая первые процессоры с ошибками.

Опции группы Syntax options управляют обработкой синтаксических конструкций Object Pascal. Опция Strict var-strings (эквивалент директивы `{SV}`) устанавлива-

Рис. 14.17.

Страница Pascal окна опций проекта



ет проверку типов коротких строк, передаваемых в функции и процедуры. Опция требуется только для обратной совместимости с ранними версиями C++Builder. При включенной опции Open parameters опция Strict var-strings не работает для открытых параметров.

Опция Complete boolean eval (эквивалент директивы **{B}**) определяет вычисление всех элементов булева выражения, даже если после вычисления первых элементов ясен результат всего выражения — **true** или **false**.

Опция Extended syntax (эквивалент директивы **{X}**) разрешает использовать функции как процедуры, игнорируя возвращаемый ими результат, а также обеспечивает поддержку типа **Pchar**.

Опция Typed @ operator (эквивалент директивы **{T}**) управляет типом указателей, возвращаемых операцией **@**.

Опция Open parameters (эквивалент директивы **{P}**) разрешает передачу открытых строк в качестве параметров процедур и функций. Опция Huge strings (эквивалент директивы **{H}**) делает возможным использование длинных строк. При включении этой опции тип **string** эквивалентен новому типу **AnsiString**. При включенной опции тип **string** эквивалентен типу **ShortString**.

Опция Assignable typed constants (эквивалент директивы **{J}**) используется для обратной совместимости с Delphi 1, разрешая присваивания типизированным константам.

Группа опций Runtime errors определяет обработку ошибок времени выполнения. Опция Range checking (эквивалент директивы **{R}**) обеспечивает проверку удовлетворения индексов массивов и строк заданным пределам. Опция I/O checking (эквивалент директивы **{I}**) обеспечивает проверку каждой операции ввода/вывода. Опция Overflow checking (Q) (эквивалент директивы **{Q}**) обеспечивает проверку переполнения при целочисленных вычислениях.

Группа опций Debugging определяет тип отладочной информации, включаемой в объектные модули **.obj**. После завершения отладки перед компиляцией законченного приложения все эти опции должны быть выключены. Опция Debug information (эквивалент директивы **{D}**) помещает отладочную информацию в файлы модулей **.dcu**. Опция Local symbols (эквивалент директивы **{L}**) генерирует информацию о локальных символах. Опция Reference info (Y) (эквивалент директивы **{Y}**) генерирует информацию о символах. Вспомогательная опция Definition only, доступная только при включенной опции Reference info (Y), определяет генерацию информации только об определениях символов. Опция Assertions (C) (эквивалент директивы **{C}**) управляет генерацией информации, связанной с процедурой **Assert**.

Группа опций Messages определяет уровень сообщений, генерируемых компилятором Object Pascal. Опция No (эквивалент директивы `{ $R }`) определяет отсутствие сообщений. Опция Show hints обеспечивает генерацию замечаний, а опция Show warnings — генерацию предупреждений.

14.2.9.7 Страница Tasm

Эта страница устанавливает опции компилятора турбо ассемблера. Операторы турбо ассемблера могут помещаться в любом месте текста и должны предваряться ключевым словом **asm**. Например,

```
asm mov ax, 0x0e07
```

или

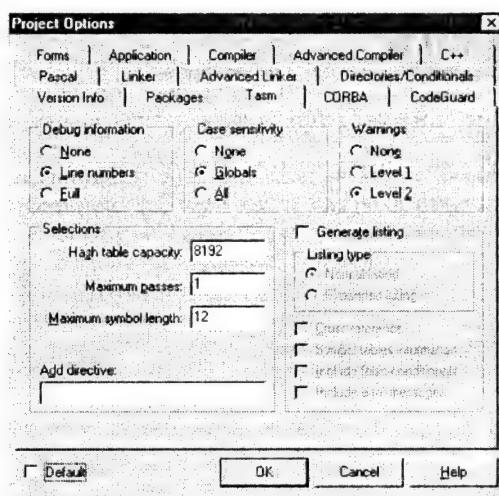
```
asm
{
mov ax, 0x0e07
xor bx, bx
int 0x10
}
```

Кроме того могут использоваться файлы **.asm**, целиком написанные на турбо ассемблере.

Общий вид страницы Tasm показан на рис. 14.18. Опции на ней скомпонованы в несколько групп.

Рис. 14.18.

Страница Tasm окна опций проекта



Опции группы Debug Information управляют включением отладочной информации в объектный модуль. Опция None запрещает включение отладочной информации. Опция Line Number включает номера строк, что позволяет синхронизовать исходный код и информацию о данных. Опция Full включает все возможности отладчика для выполнения вашей программы по шагам или изменения данных.

Опции группы Case sensitivity управляют чувствительностью к регистру. Опция None делает компилятор нечувствительным к регистру. Опция Globals обеспечивает чувствительность к регистру только внешних и открытых символов (**external** и **public**). Опция All трактует символы во всем тексте как приведенные к верхнему регистру.

Группа опций Selections определяет инициализацию и область действия ассемблера. Окно Hash table capacity задает максимально допустимое число символов файла **.asm**. Это значение может изменяться в пределах 8192 – 32768. Опция Maximum

passes устанавливает максимальное число проходов компилятора. Оно используется, если вы хотите, чтобы компилятор удалял команды **NOP**, появляющиеся при предварительных ссылках. Окно Maximum symbol length устанавливает максимальную длину значимых символов идентификаторов (должно быть не менее 12). Окно Add directive может содержать начальные директивы ассемблера, появляющиеся перед первой строкой текста файла.

Группа опций Warnings устанавливает уровень появляющихся предупреждений компилятора. Опция None исключает отображение замечаний. Опция Level 1 приводит к появлению кратких предупреждений, которые показывают, как можно повысить эффективность кода. Опция Level 2 приводит к генерации всех предупреждений.

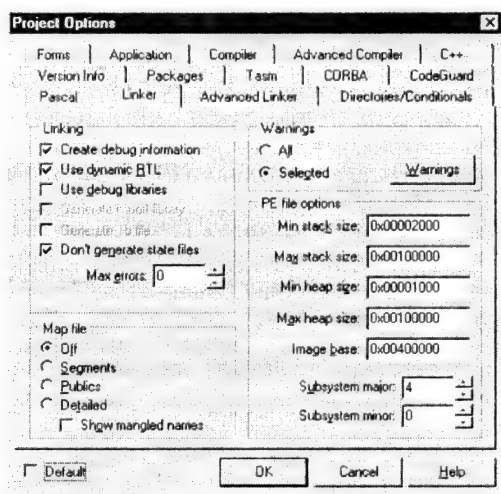
Опция Generate Listing вызывает генерацию листинга — файла с расширением **.lst**. Если эта опция включена, то остальные опции группы определяют, что именно будет включаться в файл листинга.

14.2.9.8 Страница Linker

Общий вид страницы Linker показан на рис. 14.19. Эта страница устанавливает опции, управляющие компоновкой проекта, т.е. объединением файлов **.obj**, **.lib**, **.res** в выполняемый файл **.exe** или в библиотечный файл **.dll**. Компоновка осуществляется пошаговым компоновщиком **ILINK32**. В большинстве случаев имеет смысл оставлять значения опций данной страницы теми, которые заданы по умолчанию.

Рис. 14.19.

Страница Linker окна опций проекта



Опции на странице Linker скомпонованы в несколько групп. Индикаторы группы Linking задают основные опции компоновки. Опция Create debug information добавляет в выполняемый модуль отладочную информацию, используемую как интегрированным отладчиком, так и 32 битным отладчиком TD32.EXE — Turbo Debugger.

Опция Use dynamic RTL означает использование в приложении библиотеки времени выполнения RTL в виде DLL. Если включить эту опцию, компоновщик не присоединяет RTL к вашему выполняемому файлу. В результате выполняемый файл становится существенно меньше, но ваше приложение должно передаваться другим пользователям вместе с RTL DLL. Эту опцию имеет смысл поддерживать включенной в процессе отладки приложения. Но, если вы хотите создать автономный выполняемый файл, то после завершения отладки надо перекомпоновать проект с выключенной опцией Use dynamic RTL.

Опция `Use debug libraries` означает компоновку в приложение варианта `vcl.lib` библиотеки визуальных компонентов VCL, допускающего отладку. Эта опция не работает, если вы используете поддержку пакетов. Так что для использования этой опции вы должны выключить опцию `Use Packages` на странице `Packages`. Опция `Use debug libraries` существенно увеличивает размер выполняемого модуля. Так что включать ее имеет смысл в тех редких случаях, когда вам требуется пройти по шагам в исходном коде библиотеки VCL.

Опция `Generate import library` доступна только при разработке DLL или пакета. Она управляет созданием файла импорта библиотеки `.lib` (для DLL) или пакета `.bpi` (для пакета). Этот файл необходим для использования в дальнейшем этой библиотеки или пакета. Например, если приложение использует созданную вами библиотеку, в него надо включить соответствующий файл `.lib`. А при использовании компонента из установленного пакета должен быть доступен соответствующий файл `.bpi`.

Опция `Generate .lib file` доступна только при разработке пакета. Она говорит о необходимости создания файла импорта `.lib`, а не `.bpi`. Это позволит приложению, использующему данный пакет времени выполнения, компоновать его как DLL.

Опция `Don't generate state files` запрещает генерацию файлов выборочной компоновки `.il?` с информацией о произведенной компоновке (см. раздел 14.2.9.1), в результате чего вы выиграете в дисковом пространстве, но все последующие компиляции будут проводиться так же долго, как первая. Файлы выборочной компоновки `.il?` создаются в каталоге, заданном опцией `Final output` на странице `Directories/Conditionals`.

Окно `Max errors` указывает, после скольких обнаруженных ошибок компоновка должна прекратиться. Количество ошибок можно задать до 255. При досрочном прекращении компоновки вы можете не получить сообщения о всех ошибках.

Группа опций `Map file` управляет созданием файла карты приложения `.map`, который располагается в каталоге, заданном опцией `Final output` на странице `Directories/Conditionals`. Опция `Off` исключает генерацию файла. Опция `Segments` обеспечивает включение в файл только списка сегментов, начального адреса программы и сообщений об ошибках компоновки. Опция `Publics` обеспечивает включение в файл помимо этого алфавитного списка открытых (**public**) символов. Опция `Detailed` помимо всего этого обеспечивает включение в файл детальной карты сегментов, включая адреса сегментов, их длину в байтах, имена, информацию о группах и модулях. Индикатор `Show Mangled Names` помещает в файл свернутые имена идентификаторов C++, а не полные имена. Такие имена используются в некоторых утилитах.

Группа опций `Warnings` устанавливает тип замечаний, выдаваемых при построении проекта. Опция `All` предусматривает выдачу любых типов замечаний. Опция `Selected` позволяет с помощью кнопки `Warnings` выбрать в списке те сообщения, которые желательно выдавать.

Группа окон редактирования `PE file options` указывает минимальный и максимальный размеры стека, хранящего локальные переменные и точки вызова и возврата функций (`Min stack size` и `Max stack size`), а также минимальный и максимальный размеры динамически распределяемой области памяти `heap` (`Min heap size` и `Max heap size`). Размеры задаются шестнадцатеричными цифрами. Минимальный размер стека — 4К (0x1000). По умолчанию — 0x00002000.

Окно редактирования `Image base` указывает выполняемому модулю адрес первого объекта приложения. Все объекты выравниваются по границам 64К. Эта опция уменьшает размер файла, ускоряет его загрузку и улучшает производительность. Не рекомендуется использовать эту опцию при разработке DLL. Для системы Win32 рекомендуемое значение 0x400000. Но его не надо использовать для Win32s.

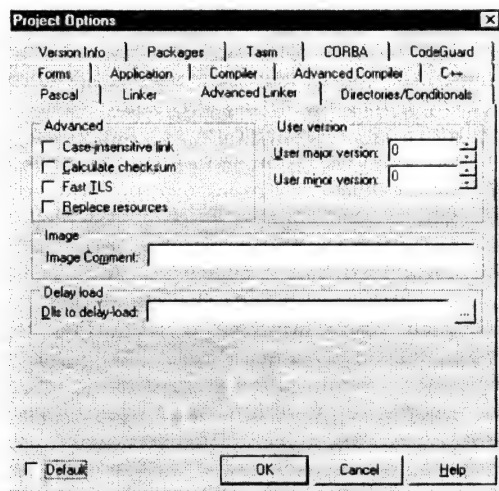
Окна Subsystem Major и Subsystem Minor позволяют указать номер версии среды разработки, для которой предназначено созданное приложение. Эта информация помещается в заголовок файла .exe. Например, для версии 4.2 следует задать Subsystem Major равным 4 и Subsystem Minor равным 2.

14.2.9.9 Страница Advanced Linker

Страница Advanced Linker (рис. 14.20) содержит дополнительные опции компоновщика. Группа опций Advanced определяет некоторые условия компоновки. Установка опции Case-insensitive link определяет чувствительность компоновщика к регистру, который используется для открытых и внешних символов. Поскольку и C, и C++ чувствительны к регистру, эту опцию, как правило, надо включать, но по умолчанию она выключена. Опция Calculate checksum задает вычисление контрольной суммы и размещение ее в результирующем файле. Это нередко требуется для драйверов и DLL. Опция Replace resources обеспечивает добавление или замещение ресурсов в выполняемом файле или DLL.

Рис. 14.20.

Страница Advanced Linker окна опций проекта



Опции User major version, User minor version и окошко Image comment позволяют занести в двоичный результирующий файл соответственно старшую и младшую цифры версии пользователя и комментариев.

В окошко Dlls to delay load можно занести список имен DLL, разделяемых точками с запятой, которые компоновщик не загружает, пока не встречается вызов какой-то их функции.

14.2.9.10 Страница Directories/Conditionals

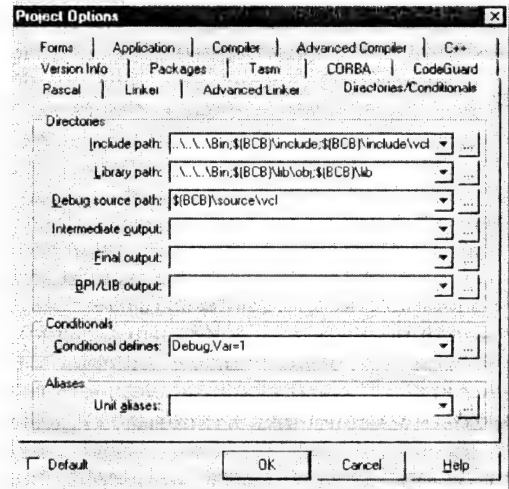
Общий вид страницы Directories/Conditionals показан на рис. 14.21. Эта страница определяет месторасположение в компьютере файлов, используемых в процессе компиляции и компоновки вашего приложения. Указываются также некоторые данные, используемые при компиляции. Около каждого окна редактирования имеется кнопка со стрелочкой. Щелчок на ней позволяет осуществить выбор из выпадающего списка ранее внесенных данных.

Окна группы Directories определяют каталоги различных файлов. Окно Include path указывает каталоги, где располагаются заголовочные файлы, используемые в проекте.

Окно Library path определяет каталоги, содержащие файлы библиотек, ресурсов и исходные файлы модулей.

Рис. 14.21.

Страница Directories/Conditionals
окна опций проекта



Окно Debug source path указывает каталоги, где отладчик может найти файлы исходных текстов. В этом окне надо указать все каталоги, в которых расположены тексты, компилирующиеся в вашем проекте или группе проектов.

Окно Intermediate output определяет каталог, в котором должны располагаться промежуточные файлы: объектные файлы **.obj** и сгенерированные файлы **.asm**.

Окно Final output указывает каталог, в который должен быть помещен результат — файл **.exe**, или **.bpl**, или **.dll**. Если этот каталог не указан, то результат размещается в том же каталоге, в котором расположен файл проекта **.bpr**.

Окно BPI/LIB output определяет при создании пакетов или DLL каталог размещения файлов **.bpi** или **.lib**. Это окно используется, если генерируются файлы импорта, т.е. если на странице Linker установлена опция Generate import library или Generate .lib file. Если окно BPI/LIB output не заполнено, то в качестве каталога используется каталог библиотек, заданный на странице Library окна Environment Options, которое открывается командой Tools | Environment Options.

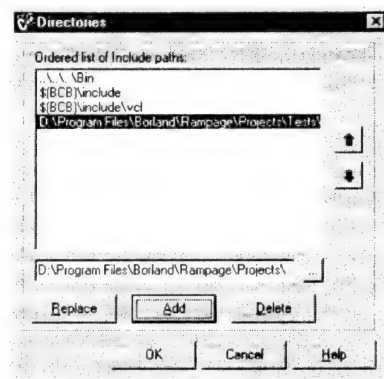
В тех из перечисленных окон, в которых может быть указано несколько каталогов, эти каталоги перечисляются, разделяясь точками с запятой. Пробелы после точек с запятой не требуются. Длина текста не должна превышать 127 символов, включая пробелы. Каждый путь может быть указан полностью, или относительно корневого каталога C++Builder с помощью макроса **\$(BCB)**.

Требуемые списки каталогов можно редактировать непосредственно в соответствующем окне. Но для первых трех окон это удобнее делать с помощью кнопок с многоточием справа от окна редактирования. Нажав эту кнопку, вы попадете в диалоговое окно, показанное на рис. 14.22. Перемещаясь по списку каталогов вы можете удалить кнопкой Delete любую строку. Если вы хотите добавить новую строку или изменить прежнюю, то можете ввести новый каталог в нижнее окно редактирования. Если при этом вам надо посмотреть дерево каталогов вашего компьютера, нажмите кнопку с многоточием около этого окна. Откроется обычный для Windows диалог, в котором вы можете выбрать каталог и нажать после этого OK. Тогда вы вернетесь в окно на рис. 14.22, в котором станут доступны кнопки Replace и Add. Первая из них заменит выделенный в верхнем окне каталог на введенный в нижнее окно, а вторая добавит новый каталог к списку. Нажав после всех этих операций кнопку OK, вы вернетесь в исходное окно рис. 14.21, в котором появится отредактированная вами строка.

Окно Conditionals страницы Directories/Conditionals (рис. 14.21) определяет условия, используемые препроцессором при условной компиляции кодов с директивами **#ifdef**, **#if** и др.. Если вам надо просто определить некоторый идентификатор,

Рис. 14.22.

Диалоговое окно списка каталогов



чтобы проверять его, например, директивой **#ifdef**, достаточно написать этот идентификатор в окне Conditionals. Если вам надо присвоить идентификатору некоторое значение, которое будет, например, проверяться директивой **#if**, то соответствующее значение присваивается символом равенства «=». Отдельные определения и присваивания разделяются запятыми. В примере на рис. 14.21 Определен идентификатор **Debug**, а идентификатору **Var** присвоено значение 1. Так что в тексте приложения сработают директивы

```
#ifdef Debug
```

```
...
```

```
и
```

```
#if Var==1
```

```
...
```

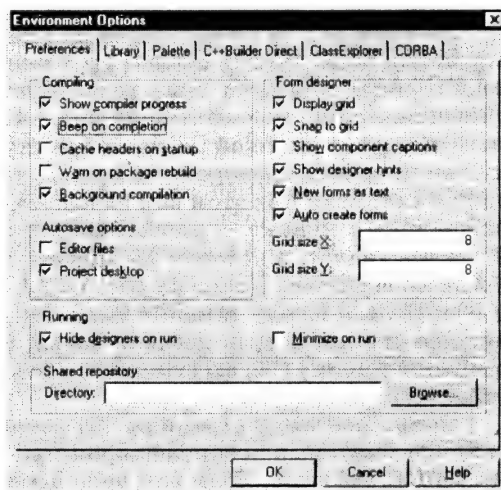
Окно Aliases страницы Directories/Conditionals позволяет задать псевдонимы для модулей Object Pascal, имена которых изменены. Синтаксис задания псевдонима: <прежнее имя>=<новое имя>.

14.2.10 Общие настройки среды

Многостраничное окно настройки вызывается командой Tools | Environment Options. На рис. 14.23 приведена его страница Preferences — общие настройки среды проектирования.

Рис. 14.23.

Страница Preferences окна Environment Options



Опции на странице Preferences скомпонованы в несколько групп. Группа Compiling определяет выполнение компиляции и компоновки. Опция Show Compiler Progress отображает диалоговое окно хода выполнения компиляции. Опция Beep on completion осуществляет звуковой сигнал по окончании компиляции. Опция Cache headers on startup обеспечивает предварительную компиляцию в оперативной памяти заголовочных файлов, что сокращает затраты времени на последующие компиляции (см. раздел 14.2.9.1). Опция Warn on package rebuild приводит к появлению предупреждения, если во время компиляции происходит перестроение пакетов. Опция Background Compilation задает выполнение всех команд компиляции (кроме Run) в фоновом режиме. Достоинства и недостатки этого обсуждаются в разделе 2.6.1.

Опции группы Autosave options определяют, что сохраняется при выполнении приложения и выходе из C++Builder. Включение опции Editor Files (редактируемые файлы) приводит к тому, что при каждом выполнении приложения и при выходе из C++Builder автоматически сохраняются все модифицированные в Редакторе Кода файлы. Это иногда может быть опасным, так как могут сохраняться помимо вашего желания какие-то неудачные исправления в программе, которые после тестирования вы хотели бы отменить. С другой стороны, это гарантирует сохранение файлов в случае, если ваше еще не отлаженное приложение при его выполнении приведет к сбою системы.

Если вы включаете опцию Project desktop (состояние экрана), то при очередном запуске C++Builder загрузится ваше последнее приложение и откроются все окна, которые были открыты в момент выхода из C++Builder. Это очень удобно, если вы намерены продолжать работу над тем же приложением.

Опции проектирования формы Form designer определяют сетку формы и ярлычки компонентов. Опция Display grid делает видимыми узлы сетки на форме. Установка опции Snap to grid приводит к тому, что компоненты при их размещении автоматически привязываются к узлам сетки. При этом компоненты невозможно разместить между узлами.

Окна редактирования Grid size x и Grid size y определяют шаг сетки по горизонтали и вертикали. Шаг может задаваться в пределах от 2 до 128.

Опция Show component captions делает видимыми надписи компонентов. Опция Show designer hints делает видимыми ярлычки с именами, типом и размерами компонентов.

Опция New forms as text определяет, в каком виде — текстовом (если опция установлена) или двоичном сохраняется файл описания формы **.dfm**. Текстовый формат легче может модифицироваться различным инструментарием, в частности, системой управления версиями. Зато двоичный формат имеет лучшую обратную совместимость с более ранними версиями C++Builder.

Опция AutoCreate Forms определяет, будут ли новые формы проекта (кроме первой) рассматриваться как автоматически создаваемые (Auto Create), или как возможные (Available Forms). В дальнейшем это можно изменить на странице Forms опций проекта.

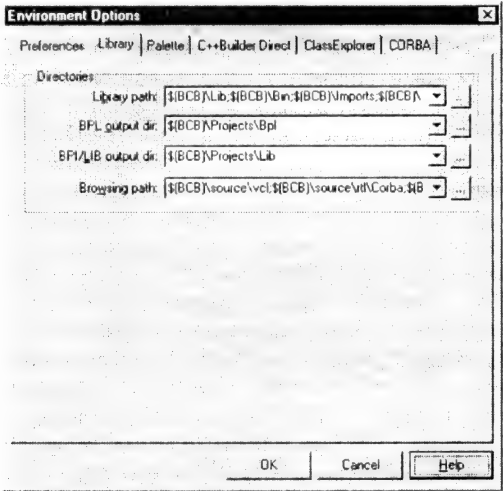
Опции группы Running определяют состояние ИСР C++Builder в процессе выполнения приложения. Установка опции Minimize on run приводит к свертыванию C++Builder при выполнении приложения. Когда приложение закрывается, C++Builder восстанавливается. Опция Hide designers on run делает невидимыми окна проектирования (Инспектора Объектов и формы) при выполнении приложения.

Окно Directory определяет местонахождение файла депозитария объектов. Если каталог не указан, C++Builder ищет этот файл в своем каталоге BIN.

Страница Library окна Environment Options (рис. 14.24) содержит списки каталогов, в которых ищутся файлы, используемые в проектах. Если указывается несколько каталогов, то они перечисляются, разделяясь точками с запятой. Пробелы после точек с запятой не требуются. Длина текста не должна превышать 127 сим-

Рис. 14.24.

Страница Library в окне настройки среды разработки



волов, включая пробелы. Каждый каталог может быть указан с полным путем, или относительно корневого каталога C++Builder с помощью макроса \$(BCB).

Кнопки со стрелками справа от окон редактирования позволяют выбрать каталог из числа ранее записанных в эти окна.

Требуемые списки каталогов можно редактировать непосредственно в соответствующем окне. Но для первого и четвертого окна, в которые можно заносить несколько каталогов, это удобнее делать с помощью кнопок с многоточием. Нажав эту кнопку, вы попадете в диалоговое окно, описанное ранее в разделе 7.9.10 и показанное на рис. 7.22.

Каталоги, указываемые в окнах страницы Library, означают следующее:

Library path	Каталоги, в которых расположены заголовочные и библиотечные файлы компонентов и пакетов. В набор этих каталогов следует включать: Каталог Include Path заголовочных файлов, используемых библиотекой VCL Каталог Library Path объектных файлов VCL, файлов ресурсов, файлов модулей C++Builder Каталоги всех библиотек, которые вы используете в своих проектах
BPL output directory	Каталог, в который компилятор должен поместить файл .bpl компилируемого пакета
BPI/LIB output path	Каталог, в котором по умолчанию должны размещаться файлы пакетов .bpi и .lib . Это значение каталога по умолчанию в дальнейшем может быть изменено для каждого пакета его опциями
Browsing path	Каталоги, в которых ищутся файлы, содержащие идентификаторы проекта

14.3. Страницы библиотеки компонентов




В этом разделе приведены те страницы библиотеки, компоненты которых рассматриваются или упоминаются в данной книге.

14.3.1 Страница Standard



Страница Standard содержит ряд часто используемых компонентов общего назначения.


Компонент	Тип	Описание
 фрейм	Frame	Панель с возможностями наследования. Проектируется как отдельное окно. Компонент визуальный.
 главное меню	MainMenu	Позволяет конструировать и создавать полосу главного меню формы и выпадающие меню. Компонент невизуальный.
 всплывающее меню	PopupMenu	Позволяет конструировать и создавать всплывающие контекстные меню, возникающие при нажатии пользователем правой кнопки мыши. Компонент невизуальный.
 метка	Label	Используется для размещения на формах и других контейнерах текста, который не изменяется пользователем. Компонент визуальный.
 окно редактирования	Edit	Используется для ввода пользователем однострочных текстов. Может использоваться для отображения текста. Компонент визуальный.
 многострочное окно редактирования	Memo	Используется для ввода и отображения многострочных текстов. Компонент визуальный.
 командная кнопка	Button	Используется для создания кнопок, которыми пользователь выбирает команды в приложении. Компонент визуальный.
 контрольный индикатор с флажком	Checkbox	Позволяет пользователю включать и выключать различные опции. Компонент визуальный.
 радиокнопка	RadioButton	Предлагают пользователю набор альтернатив, из которых выбирается одна. Набор реализуется требуемым количеством радиокнопок, размещенных в одном контейнере (форме, панели и т.п.). Компонент визуальный.

Компонент	Тип	Описание
 окно списка	ListBox	Представляет собой стандартное окно списка Windows, позволяющее пользователю выбирать разделы из списка. Компонент визуальный.
 редактируемый список	ComboBox	Объединяет функции ListBox и Edit . Пользователь может либо ввести текст, либо выбрать его из списка. Компонент визуальный.
 линейка прокрутки	Scrollbar	Представляет собой стандартную линейку прокрутки Windows и служит для управления положением видимой части форм или компонентов. Компонент визуальный.
 групповое окно	GroupBox	Является контейнером, объединяющим группу связанных органов управления, таких как радиокнопки RadioButton , контрольные индикаторы Checkbox и т.д. Компонент визуальный.
 группа радиокнопок	RadioGroup	Является комбинацией группового окна GroupBox с набором радиокнопок RadioButton ; служит специально для создания групп радиокнопок. Можно размещать в компоненте несколько радиокнопок, но никакие другие органы управления не разрешены. Компонент визуальный.
 панель	Panel	Является контейнером для группирования органов управления и меньших контейнеров. Панель можно использовать также для построения полос состояния, инструментальных панелей, палитр инструментов. Компонент визуальный.
 список событий	ActionList	Обеспечивает диспетчеризацию событий компонентов. Компонент не визуальный.

14.3.2 Страница Additional



Страница является дополнением страницы *Standard* и содержит ряд часто используемых компонентов общего назначения

Компонент	Тип	Описание
 кнопка с графикой	BitBtn	Используется для создания кнопок, на которых располагается битовая графика (например, кнопка OK с галочкой). Компонент визуальный.






Компонент	Тип	Описание
 кнопка с фиксацией	SpeedButton	Используется для создания инструментальных панелей и в других случаях, когда требуется кнопка с фиксацией нажатого состояния. Компонент визуальный.
 маскированный ввод	MaskEdit	Используется для форматирования данных или для ввода символов в соответствии с шаблоном. Компонент визуальный.
 таблица строк	StringGrid	Используется для отображения текстовой информации в таблице из строк и столбцов. Компонент визуальный.
 таблица рисунков	DrawGrid	Используется для отображения в строках и столбцах нетекстовых данных. Компонент визуальный.
 изображение	Image	Используется для отображения графики: пиктограмм, битовых матриц и метафайлов. Компонент визуальный.
 формы	Shape	Используется для рисования фигур: квадратов, кругов и т.п. Компонент визуальный.
 рамка	Bevel	Используется для рисования выступающих или утопленных линий или прямоугольных рамок. Компонент визуальный.
 окно с прокруткой	ScrollBar	Используется для создания зон отображения с прокруткой. Компонент визуальный.
 список с флажками	CheckBoxList-Box	Компонент является комбинацией свойств списка ListBox и индикаторов CheckBox в одном компоненте. Компонент визуальный.
 разделитель панелей	Splitter	Используется для создания в приложении панелей с изменяемыми пользователем размерами. Компонент визуальный.
 метка с бордюром	StaticText	Компонент подобен компоненту Label , но обеспечивает дополнительные возможности по заданию стиля бордюра. Компонент визуальный.
 инструментальная панель	ControlBar	Используется для размещения компонентов инструментальной панели. Компонент визуальный.
 события приложения	Application-Events	Перехватывает события на уровне приложения. Компонент невидимый.
 диаграммы и графики	Chart	Компонент принадлежит к семейству компонентов TChart , которые используются для создания диаграмм и графиков. Компонент визуальный.

14.3.3 Страница Win32



Страница Win32 содержит компоненты общего назначения, позволяющие разрабатывать приложения в стиле Windows 95/98/2000 и NT 4.x. Некоторые из этих компонентов аналогичны имеющимся на странице Win3.1.

Компонент	Тип	Описание
 страница с закладкой	TabControl	Позволяет организовывать страницы с закладками в стиле Windows, которые может выбирать пользователь. Компонент визуальный.
 многостраничное окно	PageControl	Позволяет создавать страницы в стиле Windows, управляемые закладками или иными органами управления, для экономии места на рабочем столе. Компонент визуальный.
 список изображений	ImageList	Предназначен для работы со списками изображений одинакового размера в меню, инструментальных панелях и т.п. Компонент не визуальный.
 окно редактирования в формате RTF	RichEdit	Представляет собой окно редактирования в стиле Windows, позволяющее производить выбор цвета и шрифта, поиск текста и многое другое. Компонент визуальный.
 ползунок	TrackBar	Управляющий элемент в виде ползунка в стиле Windows. Компонент визуальный.
 отображение хода процесса	ProgressBar	Используется для отображения в стиле Windows хода процессов, занимающих заметное время. Компонент визуальный.
 кнопка-счетчик	UpDown	Кнопка-счетчик в стиле Windows для ввода целых чисел. Компонент визуальный.
 «горячие» клавиши	HotKey	Дает возможность реализовать в приложении поддержку горячих клавиш. Компонент визуальный.
 воспроизведение немых клипов	Animate	Используется для воспроизведения немых клипов AVI, подобных используемым в Windows изображениям копирования файлов и т.п. Компонент визуальный.
 ввод дат и времени	DateTime-Picker	Ввод дат и времени с выпадающим календарем. Компонент визуальный.
 ввод дат	MonthCalendar	Ввод дат с выбором из календаря. Компонент визуальный.
 дерево	TreeView	Предоставляет возможность просмотра структуры иерархических данных в стиле Windows. Компонент визуальный.




Компонент	Тип	Описание
 списки	ListView	Отображает списки в стиле Windows. Компонент визуальный.
 заголовок	HeaderControl	Позволяет создавать составные перемещаемые заголовки в стиле Windows. Компонент визуальный.
 полоса состояния	StatusBar	Полоса состояния программы, при необходимости — на нескольких панелях. Компонент визуальный.
 инструментальная панель	ToolBar	Инструментальная панель для быстрого доступа к часто используемым функциям приложения. Компонент визуальный.
 инструментальная перестраиваемая панель	CoolBar	Контейнер инструментальной панели, размеры которой могут изменяться пользователем. Компонент визуальный.
 прокрутка страниц	PageScroller	Обеспечивает прокрутку больших окон, например, инструментальных панелей. Компонент визуальный.

14.3.4 Страница System



Страница System содержит компоненты, позволяющие использовать системные средства Windows.

Компонент	Тип	Описание
 таймер	Timer	Используется для запуска процедур, функций и событий в указанные интервалы времени. Компонент невидимый.
 окно для рисования	PaintBox	Используется для создания на форме некоторой области, в которой можно рисовать. Компонент визуальный.
 аудио и видео плеер	MediaPlayer	Используется для создания панели управления воспроизведением звуковых и видео файлов, а также устройств мультимедиа. Компонент визуальный.
 контейнер OLE	OLEContainer	Используется при создании области клиента для объекта OLE. Компонент визуальный.
 диалог с сервером DDE	DDEClientConv	Используется клиентом DDE для организации диалога с сервером DDE. Компонент невидимый.



Компонент	Тип	Описание
 данные, передаваемые серверу DDE	DDEClientItem	Используется для определения данных клиента, передаваемых в диалоге серверу DDE. Компонент невидимый.
 диалог с клиентом DDE	DDEServerConv	Компонент используется сервером DDE при проведении диалога с клиентом DDE. Компонент невидимый.
 данные, передаваемые клиенту DDE	DDEServerItem	Компонент используется для определения данных сервера, передаваемых клиенту DDE в течение диалога. Компонент невидимый.

14.3.5 Страница Data Access

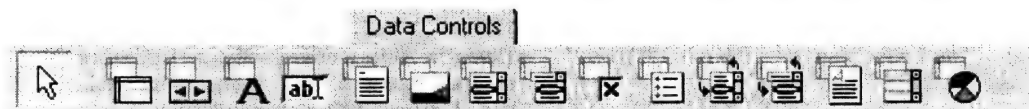


Страница Data Access — доступ к данным, содержит компоненты управления обменом информацией между приложением и базами данных.

Компонент	Тип	Описание
 источник данных	DataSource	Используется для соединения компонентов типа Table или Query с компонентами, отображающими данные. Компонент невидимый.
 набор данных	Table	Используется для установления связи приложения с таблицей базы данных. Компонент невидимый.
 запросы SQL	Query	Используется для построения и выполнения SQL-запросов к удаленным SQL-серверам или локальным базам данных. Компонент невидимый.
 хранимые процедуры	StoredProc	Используется для выполнения процедур, хранимых на SQL-сервере. Компонент невидимый.
 связь с сервером	Database	Используется для установления связи с удаленными серверами баз данных. Компонент невидимый.
 сеанс работы с данными	Session	Используется для обеспечения глобального управления соединениями приложения с базами данных. Компонент невидимый.
 локальная работа с данными	BatchMove	Позволяет работать с записями и таблицами локально, а затем передавать обновленную информацию обратно на сервер. Компонент невидимый.

Компонент	Тип	Описание
 изменение базы данных SQL	UpdateSQL	Используется для внесения изменений в базы данных SQL. Компонент не визуальный.
 вложенные таблицы	NestedTable	Таблицы данных, вложенные в другие таблицы как поля. Компонент не визуальный.

14.3.6 Страница Data Controls

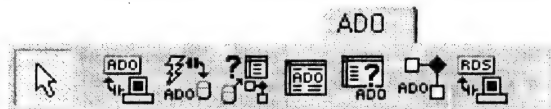


Страница Data Controls содержит компоненты, связанные с данными и предназначенные для отображения, ввода и редактирования содержимого баз данных.



Компонент	Тип	Описание
 таблица данных	DBGrid	Используется для создания ориентированных на данные таблиц с отображением данных в строках и столбцах. Компонент визуальный.
 навигатор	DBNavigator	Используется для управления просмотром и редактированием базы данных. Компонент визуальный.
 метка	DBText	Представляет собой ориентированный на данные вариант метки Label . Компонент визуальный.
 окно редактирования	DBEdit	Представляет собой ориентированный на данные вариант окна редактирования Edit . Компонент визуальный.
 окно редактирования Memo	DBMemo	Представляет собой ориентированный на данные вариант многострочного окна редактирования Memo . Компонент визуальный.
 изображение	DBImage	Представляет собой ориентированный на данные вариант компонента Image . Компонент визуальный.
 список	DBListBox	Представляет собой ориентированный на данные вариант компонента List-Box . Компонент визуальный.
 редактируемый список	DBComboBox	Представляет собой ориентированный на данные вариант компонента ComboBox . Компонент визуальный.
 индикатор	DBCheckBox	Представляет собой ориентированный на данные вариант компонента CheckBox . Компонент визуальный.






Компонент	Тип	Описание
 группа радиокнопок	DBRadioGroup	Представляет собой ориентированный на данные вариант компонента RadioGroup . Компонент визуальный.
 создание списка	DBLookupList-Box	Предназначен для создания окна List-Box , ориентированного на данные. Компонент визуальный.
 создание редактируемого списка	DBLookupCombo-Box	Предназначен для создания окна ComboBox , ориентированного на данные. Компонент визуальный.
 окно формата RTF	DBRichEdit	Представляет собой ориентированный на данные вариант компонента RichEdit . Компонент визуальный.
 гибкая таблица данных	DBCtrlGrid	Используется для создания таблицы данных, более гибкой, чем DBGrid . Компонент визуальный.
 гистограммы, графики	DBChart	Представляет собой ориентированный на данные вариант компонента Chart . Компонент визуальный.

14.3.7 Страница ADO

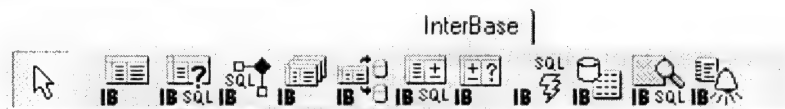


Страница ADO, появившаяся только в C++Builder 5, содержит компоненты для связи с базами данных через Active Data Objects (ADO) — множество компонентов ActiveX, использующих для доступа к информации баз данных Microsoft OLE DB. Эти компоненты являются альтернативой компонентам, использующим BDE и размещенным на странице Data Access.






Компонент	Тип	Описание
 соединение	ADOConnection	Используется для связи с набором данных ADO. Может работать с несколькими компонентами наборов данных как диспетчер выполнение их команд. Компонент невидимый.
 команды SQL	ADOCommand	Используется в основном для выполнения команд SQL, не возвращающих множество результатов. Может также совместно с другими компонентами использоваться для работы с таблицами. Может связываться с набором данных непосредственно, или через ADOConnection . Компонент невидимый.

Компонент	Тип	Описание
 универсальный набор данных	ADODataSet	Универсальный компонент связи с наборами данных, который может работать в различных режимах, заменяя связанные с BDE компоненты Table , Query , StoredProc . Компонент невидимый.
 набор данных	ADOTable	Аналог Table , используемый для работы с одной таблицей. Компонент невидимый.
 запросы SQL	ADOQuery	Используется для работы с набором данных с помощью запросов SQL. Компонент невидимый.
 хранимые процедуры	ADOSToredProc	Используется для выполнения процедур, хранимых на сервере. Компонент невидимый.
 объект RDS		Представляет объект RDS DataSpace и используется в многопоточных приложениях. Компонент невидимый.

14.3.8 Страница InterBase

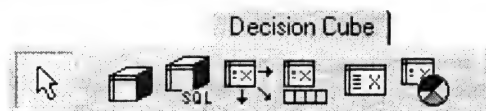


Страница InterBase, появившаяся только в C++Builder 5, содержит компоненты для работы с InterBase напрямую, минуя BDE. Эти компоненты обеспечивают повышенную производительность и позволяют использовать новые возможности сервера InterBase 5.5, недоступные обычным компонентам BDE.






Компонент	Тип	Описание
 набор данных	IBTable	Используется вместо Table для доступа к одной таблице набора данных. Компонент невидимый.
 запросы SQL	IBQuery	Используется вместо Query для выполнения запросов SQL. Компонент невидимый.
 хранимые процедуры	IBStoredProc	Используется для выполнения процедур, хранимых на сервере. Компонент невидимый.
 соединение с InterBase	IBDatabase	Обеспечивает соединение с базой данных InterBase. Компонент невидимый.
 транзакции	IBTransaction	Управляет транзакциями. Поддерживаются распределенные транзакции со множеством баз данных. Компонент невидимый.


Компонент	Тип	Описание
 изменение данных, кэширование	IBUpdateSQL	Используется для изменений в таблицах только для чтения и для кэширования изменений. Компонент невидимый.
 набор данных	IBDataSet	Набор данных, обеспечивающий эффективный доступ к данным. Компонент невидимый.
 запросы SQL	IBSQL	Выполняет запросы SQL, минимизируя затраты буферизации и обмена данными с компонентами Delphi. Обеспечивает наиболее эффективный доступ к данным InterBase. Компонент невидимый.
 информация	IBDatabase-Info	Позволяет приложению затребовать информацию о базе данных и сервере InterBase. Компонент невидимый.
 мониторинг	IBSQLMonitor	Позволяет осуществлять отладку коммуникаций для ускорения работы проектов клиент/сервер и проектов с параллельными потоками. Компонент невидимый.
 события	IBEvents	Обеспечивает асинхронную обработку событий сервера InterBase. Компонент невидимый.

14.3.9 Страница Decision Cube



Страница Decision Cube содержит компоненты для проведения многомерного анализа данных, хранимых в базах данных.

Компонент	Тип	Описание
 многомерный куб	DecisionCube	Связь с базой данных и реализация многомерного куба — основы многомерного анализа данных. Компонент невидимый.
 набор данных	DecisionQuery	Набор данных, на основании которого проводится анализ. Компонент невидимый.
 источник данных	DecisionSource	Источник многомерных данных — посредник между DecisionCube и компонентами отображения данных. Компонент невидимый.
 управление	DecisionPivot	Управление отдельными измерениями куба. Компонент визуальный.
 таблица	DecisionGrid	Табличное отображение данных. Компонент визуальный.


Компонент	Тип	Описание
 графика	DecisionGraph	Графическое отображение данных. Компонент визуальный.

14.3.10 Страница QReport



Страница QReport содержит компоненты, используемые при генерации отчетов.

Компонент	Тип	Описание
 отчет	QuickRep	Используется для введения в приложение средств печати отчетов QuickReport. Компонент невидимый.
 детали	QRSubDetail	Используется для компоновки в отчет дополнительных данных. Компонент визуальный.
 полоса текста	QRStringsBand	Используется для компоновки в отчет дополнительных текстов. Компонент визуальный.
 полоса	QRBand	Используется для построения отчетов путем размещения на нем печатаемых компонентов. Компонент визуальный.
 дочерняя полоса	QRChildBand	Используется для создания дочерних полос, которые могут содержать другие компоненты QuickRep и полосы. Компонент визуальный.
 группировка	QRGroup	Используется для группировки данных. Компонент невидимый.
 метка	QRLabel	Используется для размещения текста в отчете. Компонент визуальный.
 текст из базы данных	QRDBText	Представляет собой ориентированный на данные компонент для размещения текста в отчете. Компонент визуальный.
 математические выражения	QRExpr	Позволяет строить и отображать выражения над полями данных и системными величинами (такими, как время и дата).
 системные данные	QRSysData	Используется для отображения системных данных. Компонент визуальный.
 многострочный текст	QRMemo	Используется для размещения в отчете многострочных текстов. Компонент визуальный.

Компонент	Тип	Описание
 тексты с математическими выражениями	QRExprMemo	Используется для размещения в отчете текстов с математическими выражениями. Компонент визуальный.
 многострочный текст RTF	QRRichText	Используется для размещения в отчете текста в обогащенном формате RichText . Компонент визуальный.
 многострочный текст RTF базы данных	QRDBRichText	Используется для размещения в отчете текста из базы данных в обогащенном формате RichText . Компонент визуальный.
 форма	QRShape	Используется для рисования в отчете графических форм. Компонент визуальный.
 изображение	QRImage	Используется для печати изображений в отчете. Компонент визуальный.
 изображение из базы данных	QRDBImage	Используется для печати изображений из баз данных в отчете. Компонент визуальный.
 составной отчет	QRComposite-Report	Используется для построения составных отчетов. Компонент визуальный.
 предварительный просмотр	QRPreview	Используется для предварительного просмотра на экране подготовленного к печати отчета. Компонент визуальный.
 фильтр текста	QRTextFilter	Используется для установки фильтра текста. Компонент невидимый.
 разделитель	QRCSVFilter	Используется для установки разделителя текста. Компонент невидимый.
 фильтр HTML	QRHTMLFilter	Используется для установки фильтра HTML. Компонент невидимый.
 диаграммы, графики	QRChart	Используется для печати в отчете диаграмм, построенных на основе баз данных. Компонент визуальный.

14.3.11 Страница Dialogs





Страница Dialogs содержит компоненты, используемые для создания различных диалоговых окон, общепринятых в приложениях Windows. Диалоги используются для указания файлов или выбора установок. Применение поставляемых в составе Delphi диалоговых окон помогает сэкономить время на разработку и придать вашему приложению совместимость с принятыми в Windows нормами диалогов.

Компонент	Тип	Описание
 «Открыть файл»	OpenDialog	Предназначен для создания окна диалога «Открыть файл». Компонент невидимый.
 «Сохранить файл как...»	SaveDialog	Предназначен для создания окна диалога «Сохранить файл как». Компонент невидимый.
 «Открыть рисунок»	OpenPictureDialog	Предназначен для создания окна диалога «Открыть рисунок». Компонент невидимый.
 «Сохранить рисунок как...»	SavePictureDialog	Предназначен для создания окна диалога «Сохранить рисунок как». Компонент невидимый.
 «Шрифты»	FontDialog	Предназначен для создания окна диалога «Шрифты». Компонент невидимый.
 «Цвет»	ColorDialog	Предназначен для создания окна диалога «Цвет». Компонент невидимый.
 «Печать»	PrintDialog	Предназначен для создания окна диалога «Печать». Компонент невидимый.
 «Установка принтера»	PrinterSetupDialog	Предназначен для создания окна диалога «Установка принтера». Компонент невидимый.
 «Найти»	FindDialog	Предназначен для создания окна диалога «Найти». Компонент невидимый.
 «Заменить»	ReplaceDialog	Предназначен для создания окна диалога «Заменить». Компонент невидимый.

14.3.12 Страница Win3.1



Страница Win3.1 содержит компоненты, предназначенные для приложений Windows 3x. В 32-разрядных приложениях компоненты данной страницы применять не рекомендуется.


Компонент	Тип	Описание
 блокнот с закладками	TabSet	Используется для создания блокнота с закладками. Компонент визуальный.
 окно дерева	Outline	Позволяет отображать иерархические данные в форме дерева. Компонент визуальный.



Компонент	Тип	Описание
 многостраничная форма	TabbedNoteBook	Используется для создания многостраничных форм с закладками. Компонент визуальный.
 пачка страниц	NoteBook	Используется для создания пачки страниц, может применяться совместно с TabSet . Компонент визуальный.
 заголовок	Header	Используется для отображения текста в областях переменного размера. Компонент визуальный.
 список файлов	FileListBox	Отображает список файлов каталога. Компонент визуальный.
 структура каталогов	DirectoryListBox	Отображает структуру каталогов диска. Компонент визуальный.
 список дисков	DriveComboBox	Выпадающий список доступных дисков. Компонент визуальный.
 список фильтров	FilterComboBox	Выпадающий список фильтров для поиска файлов. Компонент визуальный.
 создание списка данных	DBLookupList	Предназначен для просмотра значений в таблице данных с помощью окна списка. Компонент визуальный.
 создание редактируемого списка данных	DBLookupCombo	Предназначен для просмотра значений в таблице данных с помощью выпадающего списка. Компонент визуальный.

14.3.13 Страница Samples



Страница Samples содержит примеры компонентов. Поскольку это всего лишь примеры, они снабжены в C++Builder лишь минимальной документацией и во встроенной справке сведения о них отсутствуют. Однако, исходные тексты примеров со страницы Samples поставляются вместе с C++Builder 5. Вы можете их просмотреть и понять, как построены эти примеры и как ими пользоваться.




Компонент	Тип	Описание
 индикатор хода процесса	CGauge	Пример компонента, используемого для создания индикатора хода процесса в виде линейки, текста или секторной диаграммы. Компонент визуальный.

Компонент	Тип	Описание
 таблица цветов	CColorGrid	Пример компонента, используемого для создания таблицы цветов, в которой пользователь выбирает требуемый цвет. Компонент визуальный.
 кнопка-счетчик	CSpin-Button	Пример компонента, используемого для создания кнопок-счетчиков. Компонент визуальный.
 окно редактирования со счетчиком	CSpinEdit	Пример компонента, используемого для создания окна редактирования в комбинации с кнопкой-счетчиком. Компонент визуальный.
 дерево каталогов	CDirectory-Outline	Пример компонента, используемого для отображения структуры каталогов выбранного диска. Компонент визуальный.
 календарь	CCalendar	Пример компонента, используемого для отображения календаря на указанный месяц в стандартном формате. Компонент визуальный.
 индикатор события	IBEvent-Alerter	Пример компонента, сигнализирующего о событии в базе данных. Компонент не визуальный.

14.3.14 Страница ActiveX



Страница ActiveX содержит примеры компонентов ActiveX. Поскольку это всего лишь примеры, они снабжены в C++Builder лишь минимальной документацией и во встроенной справке сведения о них отсутствуют. Но если вы перенесете соответствующий компонент на форму и щелкнете на нем правой кнопкой мыши, то во всплывшем меню можете выбрать команду Property и некоторые другие, которые отобразят диалоговые окна, помогающие задать необходимые свойства компонента.

Компонент	Тип	Описание
 диаграммы и графики	Chartfx	Редактор диаграмм и графиков.
 орфографический контроль	VCSpell	Визуальный блок орфографического контроля
 страницы Excel	F1Book	Компонент ввода и обработки числовой информации, аналогичный страницам Excel.
 диаграммы	VtChart	Окно построения диаграмм.
 диаграммы и графики	Graph	Окно построения диаграмм и графиков.

14.3.15 Страница Servers



Страница Servers содержит множество серверов COM, соответствующих основным программам Windows. На приведенном выше рисунке показана только часть этих серверов. Компоненты этой страницы позволяют вызывать из вашего приложения такие программы Windows, как Word, Excel, Access и др.

Глава 15

Функции C, C++, библиотек C++Builder, API Windows

В настоящей главе описано свыше 570 функций C, C++, библиотек C++Builder, API Windows. Это еще далеко не все функции, которые можно использовать. Но ограничения на объем книги потребовали отобрать из всего трудно обозримого множества функций те, которые используются чаще всего.

15.1 Справочные сведения общего характера

15.1.1 Коды клавиш

Ниже приведены виртуальные коды клавиш, которыми можно пользоваться при обработке символов, строк, при проверке параметра **Key** в обработчиках событий **OnKeyDown** и **OnKeyUp**. Символы кириллицы соответствуют тем клавишам с латинскими символами, на которых они размещены. Использование виртуальных кодов клавиш см. в главе 4 в разделе 4.3.2.2.

Клавиша	Десятичное число	Шестнадцатеричное число	Символическое имя	Сравнение по символу
F1	112	0x70	VK_F1	
F2	113	0x71	VK_F2	
F3	114	0x72	VK_F3	
F4	115	0x73	VK_F4	
F5	116	0x74	VK_F5	
F6	117	0x75	VK_F6	
F7	118	0x76	VK_F7	
F8	119	0x77	VK_F8	
F9	120	0x78	VK_F9	
F10	121	0x79	VK_F10	
пробел	32	0x20	VK_SPACE	
BackSpace	8	0x8	VK_BACK	
Tab	9	0x9	VK_TAB	
Enter	13	0x0D	VK_RETURN	
Shift	16	0x10	VK_SHIFT	
Ctrl	17	0x11	VK_CONTROL	
Alt	18	0x12	VK_MENU	
CapsLock	20	0x14	VK_CAPITAL	

Клавиша	Десятичное число	Шестнадцате- ричное число	Символическое имя	Сравнение по символу
Esc	27	0x1B	VK_ESCAPE	
Insert	45	0x2D	VK_INSERT	
PageUp	33	0x21	VK_PRIOR	
PageDown	34	0x22	VK_NEXT	
End	35	0x23	VK_END	
Home	36	0x24	VK_HOME	
←	37	0x25	VK_LEFT	
↑	38	0x26	VK_UP	
→	39	0x27	VK_RIGHT	
↓	40	0x28	VK_DOWN	
Delete	46	0x2E	VK_DELETE	
PrintScreen	44	0x2C	VK_SNAPSHOT	
ScrollLock	145	0x91	VK_SCROLL	
Pause	19	0x13	VK_PAUSE	
NumLock	144	0x90	VK_NUMLOCK	
0,)	48	0x30		'0'
1 !	49	0x31		'1'
2 @	50	0x32		'2'
3 #	51	0x33		'3'
4 \$	52	0x34		'4'
5 %	53	0x35		'5'
6 ^	54	0x36		'6'
7 &	55	0x37		'7'
8 *	56	0x38		'8'
9 (57	0x39		'9'
` ~	192	0xC0		
- _	189	0xBD		
= +	187	0xBB		
[{	219	0xDB		
] }	221	0xDD		
;;	186	0xBA		
' «	222	0xDE		
\	220	0xDC		
, <	188	0xBC		
. >	190	0xBE		

Клавиша	Десятичное число	Шестнадцате- ричное число	Символическое имя	Сравнение по символу
/ ?	191	0xBF		
a,A	65	0x41		'A'
b,B	66	0x42		'B'
c,C	67	0x43		'C'
d,D	68	0x44		'D'
e,E	69	0x45		'E'
f,F	70	0x46		'F'
g,G	71	0x47		'G'
h,H	72	0x48		'H'
i,I	73	0x49		'I'
j,J	74	0x4A		'J'
k,K	75	0x4B		'K'
l,L	76	0x4C		'L'
m,M	77	0x4D		'M'
n,N	78	0x4E		'N'
o,O	79	0x4F		'O'
p,P	80	0x50		'P'
q,Q	81	0x51		'Q'
r,R	82	0x52		'R'
s,S	83	0x53		'S'
t,T	84	0x54		'T'
u,U	85	0x55		'U'
v,V	86	0x56		'V'
w,W	87	0x57		'W'
x,X	88	0x58		'X'
y,Y	89	0x59		'Y'
z,Z	90	0x5A		'Z'

На правой клавиатуре при выключенной клавише NumLock

0	96	0x60	VK_NUMPAD0	
1	97	0x61	VK_NUMPAD1	
2	98	0x62	VK_NUMPAD2	
3	99	0x63	VK_NUMPAD3	
4	100	0x64	VK_NUMPAD4	
5	101	0x65	VK_NUMPAD5	
6	102	0x66	VK_NUMPAD6	

Клавиша	Десятичное число	Шестнадцатеричное число	Символическое имя	Сравнение по символу
7	103	0x67	VK_NUMPAD7	
8	104	0x68	VK_NUMPAD8	
9	105	0x69	VK_NUMPAD9	
*	106	0x6A	VK_MULTIPLY	
+	107	0x6B	VK_ADD	
-	109	0x6D	VK_SUBTRACT	
.	110	0x6E	VK_DECIMAL	
/	111	0x6F	VK_DIVIDE	

15.1.2 Некоторые объявленные константы C++Builder

В C++Builder предопределен ряд глобальных идентификаторов — макросов, называемых иногда объявленными (manifest) константами. Большинство из них начинаются с двух символов подчеркивания. В приведенной ниже таблице для большей наглядности и во избежание путаницы между этими символами подчеркивания введены пробелы, т.е. вместо (__) записано (_). В реальных идентификаторах этот пробел не должен фигурировать. Ниже приводится только часть объявленных констант. Остальные вы можете посмотреть во встроенной справке C++Builder.

Макросы `__DATE_`, `__FILE_`, `__LINE_`, `__STDC_` и `__TIME_` не должны появляться в файле непосредственно за директивами `#define` и `#undef`.

Макрос	Значение	Описание
<code>__BCOPT_</code>	1	Определен в любом компиляторе, производящем оптимизацию
<code>__BCPLUSPLUS_</code>	0x0530	Определен, если вы выбрали компиляцию C++; в последующих версиях значение будет увеличено
<code>__BORLANDC_</code>	0x0530	Номер версии
<code>__CDECL_</code>	1	Определен, если установлено соглашение вызова <code>cdecl</code>
<code>_CHAR_UNSIGNED</code>	1	Определен, если выбрана опция, что по умолчанию тип <code>char</code> эквивалентен <code>unsigned char</code> ; при опции <code>-K</code> макрос не определен
<code>__CONSOLE_</code>		Определен для консольных приложений
<code>_CPPUNWIND</code>	1	По умолчанию определен и показывает, что доступно разматывание стека; при <code>-xd</code> не определен
<code>__cplusplus</code>	1	Определен в режиме C++
<code>__DATE_</code>	строка	Дата компиляции исходного файла (строка в формате «Mmm dd ууу», например, «Jan 19 1994»)
<code>__DLL_</code>	1	Определен при использовании опции <code>-WD</code>

Макрос	Значение	Описание
<code>__FILE__</code>	строка	Предполагаемое имя исходного файла
<code>__FLAT__</code>	1	Определен при компиляции в модели памяти flat с разрядностью 32 бита
<code>__LINE__</code>	целое	Номер текущей строки исходного текста программы
<code>__PASCAL__</code>	1	Определен, если установлено соглашение вызова Pascal
<code>__STDC__</code>	1	Используется для указания, что данная реализация удовлетворяет стандартам ANSI. Определен, если вы компилировали с опцией -A
<code>__TEMPLATES__</code>	1	Определен для файлов C++, означает, что поддерживаются шаблоны
<code>__TIME__</code>	строка	Время компиляции исходного файла (символьная строка формата «hh:mm:ss»)
<code>__WIN32__</code>	1	Определен для приложений консольных и GUI

Комментарий

Приведенные объявленные константы C++Builder могут использоваться в приложениях для определения различных опций настройки приложения. Приведенный ниже оператор

```
Edit1->Text = __DATE__;
```

отображает в окне редактирования **Edit1** дату создания файла. Например, результат выполнения этого оператора может иметь вид:

```
Apr 10 2000
```

Если необходимо включить значение объявленной константы в строку текста, ее надо явным образом привести к типу строки. Например, оператор

```
Edit1->Text = "Дата создания: " + (String)__DATE__ +  
", >xDX1: " + (String)__TIME__;
```

даст результат вида:

```
Дата создания: Apr 10 2000, время: 17:28:34
```

15.1.3 Управляющие последовательности символов (escape-последовательности)

Управляющие последовательности символов состоят из символа обратного слеша (\) и одного или нескольких символов, следующих за ним без пробела.

Если за символом (\) следуют от одной до трех восьмеричных цифр или любое количество шестнадцатеричных цифр, то такая последовательность воспринимается как соответствующее восьмеричное или шестнадцатеричное число. Например: `'\03'` или `'\0xf'`.

Ниже приведена таблица основных управляющих последовательностей.

Последовательность	Значение	Описание
\a	0x07	звонок
\b	0x08	забивание символа слева
\f	0x0C	новая страница
\n	0x0A	новая строка
\r	0x0D	возврат каретки
\t	0x09	горизонтальная табуляция
\v	0x0B	вертикальная табуляция
\\	0x5C	обратный слеш
\'	0x27	одинарная кавычка
\"	0x22	двойная кавычка
\?	0x3F	вопросительный знак
\...		восьмеричное число (до трех цифр)
\x...		шестнадцатеричное число
\X...		шестнадцатеричное число

Рассмотрим примеры. При работе со строками, содержащими путь к некоторому файлу, надо не забывать, что одиночный обратный слеш воспринимается как начало управляющей последовательности. Поэтому строка вида

```
"d:\test\readmy.txt"
```

будет воспринята не как имя файла с путем, а как

```
d:<символ табуляции>est<символ возврата каретки>eadmy.txt
```

Чтобы восприятие строки было нормальным, надо удвоить все символы обратного слеша:

```
"d:\\test\\readmy.txt"
```

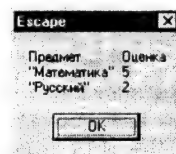
Ниже приведен пример, отображающий с помощью функции **ShowMessage** (см. раздел 15.7.2.1) окно сообщения с табулированным текстом:

```
ShowMessage (
    "Предмет\tОценка\n"Математика"\t5\n"Русский"\t2");
```

Результат выполнения этого оператора показан на рис. 15.1. Символы новой строки «\n» позволили разбить текст на три строки. Символы табуляции «\t» позволили расположить текст каждой строки в двух столбцах. А символы «\» позволили ввести в текст кавычки.

Рис. 15.1.

Пример сообщения с табулированным выводом



15.1.4 Форматы и типы, используемые при форматировании данных

15.1.4.1 Строка форматирования функций вывода

Строка формата, используемая во многих функциях вывода данных (**printf**, **cprintf**, **sprintf** и др.), состоит из обычных символов, управляющих последовательностей символов (см. раздел 15.1.3) и спецификаций полей формата вывода аргументов. Обычные символы и управляющие последовательности просто копируются в выходную строку.

Спецификации полей формата начинаются с символа **%** и имеют вид:

```
%[flags][width][.precision][F|N|h|l|L]type
```

Все символы спецификации записываются без пробелов между ними.

Единственно обязательным элементом спецификации является **type** — символ, указывающий на тип данных вводимого поля. Остальные необязательные элементы задают параметры форматирования:

[flags]	Флаги выравнивания, управления печатью знака числа, управления пробелами, десятичной точкой, основанием печати (восьмеричная, шестнадцатеричная)
[width]	Ширина поля — минимальное число выводимых символов
[.precision]	Спецификатор точности — максимальное количество печатаемых символов или минимальное количество разрядов печатаемого целого
[F N h l L]	Модификаторы, изменяющие размер аргумента по умолчанию:
	N ближний указатель (near)
	F дальний указатель (far)
	h short int
	l long
	L long double

Ниже приведены возможные значения **type**.

Символ	Тип аргумента	Формат вывода
Числа		
d	целый	десятичное signed integer
i	целый	десятичное signed integer
o	целый	восьмеричное unsigned integer
u	целый	десятичное unsigned integer
x	целый	шестнадцатеричное unsigned int (с цифрами a, b, c, d, e, f)
X	целый	то же, что x , но с цифрами A, B, C, D, E, F
f	действительный	формат с фиксированной точкой: [-]dddd.dddd
e	действительный	экспоненциальный (научный) формат: [-]d.dddd e[+/-]ddd

Символ	Тип аргумента	Формат вывода
E	действительный	то же, что e , но с символом E
g	действительный	наиболее компактный из форматов e и f для данного числа и данной точности; незначащие нули не выводятся
G	действительный	то же, что g , но с символом E в экспоненциальном формате
Символы		
c	символ	один символ
s	указатель на строку	строка символов до нулевого символа в конце или с числом символов, заданных точностью
%	нет	печать символа %
Указатели		
p	указатель на int	в ячейку памяти, на которую указывает аргумент, заносится количество выведенных к данному моменту символов
p	указатель	печать аргумента как указателя; в зависимости от используемой модели памяти печатается или XXXX:YYYY , или YYYY (только смещение)

Перечисленные ниже флаги **flags** могут записываться в любой последовательности и в любой комбинации.

Флаг	Пояснение
-	Выравнивание влево, оставшееся поле справа заполняется пробелами. Если этот флаг не задан, то производится выравнивание вправо, а оставшееся поле слева заполняется нулями или пробелами.
+	Обязательно перед числом указывается знак плюс (+) или минус (-).
пробел	Если значение не отрицательное, то печать начинается с пробела вместо знака плюс (+). Для отрицательного значения знак минус (-) печатается. Если наряду с этим флагом задан флаг +, то он должен быть указан до флага пробела.
#	В форматах o , x , X добавляется префикс 0 , 0x , 0X соответственно. В форматах e , E , f , g , G во всех случаях выводится десятичная точка. Кроме того в форматах g , G не подавляется вывод незначащих нулей.

Спецификатор **width** задает минимальную ширину поля. Спецификатор может быть задан или явным образом — десятичным числом, или косвенно — символом звездочки (*). В последнем случае предполагается, что ширину поля задает очередной аргумент из списка.

Спецификатор **width** указывает только минимальную ширину. Если вывод данного аргумента требует большей ширины поля, то поле расширяется и значение никогда не усекается.

Спецификатор может принимать следующие значения:

n	Выводится по крайней мере n символов. Если для вывода требуется меньше символов, то лишние позиции (слева или справа, в зависимости от флагов) заполняются пробелами
On	Выводится по крайней мере n символов. Если для вывода требуется меньше символов, то лишние позиции слева заполняются нулями
*	Ширину поля задает очередной аргумент из списка

Спецификатор точности **precision** определяет максимальное число выводимых символов или место десятичной точки. Он записывается после символа точки (.), чтобы отделить его от предшествующего спецификатора **width**. Данный спецификатор, как и **width**, может быть задан или явным образом — десятичным числом, или косвенно — символом звездочки (*). В последнем случае предполагается, что точность задается очередным аргументом из списка.

Отсутствие спецификатора **precision** означает точность по умолчанию и эквивалентно:

1	для форматов d, i, o, u, x, X
6	для форматов e, E, f
числу значащих цифр	для форматов g, G
выводу до нулевого символа	для формата s
не влияет	на формат c

Возможные значения **precision**:

.0	Для форматов d, i, o, u, x эквивалентно точности по умолчанию Для форматов e, E, f означает вывод без десятичной точки
.n	Задаёт вывод n символов или позицию n десятичной точки. Если выводимая величина содержит более n символов, то строка символов усекается, а число может округляться (в зависимости от формата)
*	Точность задает очередной аргумент из списка

Ниже приведены сведения о влиянии значения **precision** на различные форматы.

d, i, o, u, x, X	Указывает, что должно выводиться по крайней мере n цифр. Если число имеет менее n цифр, позиции слева заполняются нулями. Если число имеет более n цифр, число не усекается
e, E, f	Указывает, что после десятичной точки должно выводиться n цифр. Последняя цифра округляется
g, G	Указывает, что должно выводиться до n цифр
c	Спецификатор не влияет
s	Указывает, что должно выводиться не более n символов

Примеры

Примеры влияния формата:

% f	% e	% g	% #G
110000.000000	1.100000e+05	110000	110000.
-110000000.000000	-1.100000e+08	-1.1e+08	-1.10000E+08
0.000110	1.100000e-04	0.00011	0.000110000
0.000000	1.100000e-07	1.1e-07	1.10000E-07
12.000000	1.200000e+01	12	12.0000
0.000000	0.000000e+00	0	0.00000

Примеры влияние флагов:

Спецификация	Результат
% 6i	12 -12
% -6i	12 -12
% +6i	+12 -12
% 06i	000012 -00012

Примеры влияния точности:

Спецификация		
% f	123456789.000000	0.123457
% .5f	123456789.00000	0.12346
% .4f	123456789.0000	0.1235
% .3f	123456789.000	0.123
% e	1.234568e+08	1.234568e-01
% .5e	1.23457e+08	1.23457e-01
% .4e	1.2346e+08	1.2346e-01
% .3e	1.235e+08	1.235e-01
% g	1.23457e+08	0.123457
% .5g	1.2346e+08	0.12346
% .4g	1.235e+08	0.1235
% .3g	1.23e+08	0.123
% .2g	1.2e+08	0.12

Примеры использования строка форматирования вы можете также найти в разделе 15.5.4 в описании функций вывода.

15.1.4.2 Строка форматирования функций ввода

Описанная ниже строка формата, используется во многих функциях ввода данных (**scanf**, **fscanf**, **sscanf** и др.). Строка может включать три вида элементов:

- пробельные символы (пробел «», табуляцию «**\t**», символ новой строки «**\n**»
- не пробельные печатные символы (кроме **%**)
- спецификации формата

Если в строке встретился пробельный символ, то с этого момента пробельные символы до первого не пробельного символа считываются из входного потока, но не участвуют в присваивании значений переменным (игнорируются).

Если в строке встретился печатный не пробельный символ, то с этого момента из входного потока считывается и игнорируется последовательность символов, встретившаяся в строке формата. Если последовательность символов во входном потоке не соответствует записанной в строке формата, то форматирование прерывается.

Спецификации формата начинаются с символа **%** и имеют вид:

% [*****] [**width**] [**F|N**] [**h|l|L**] **type**

Все символы спецификации записываются без пробелов между ними.

Единственно обязательным элементом спецификации является **type** — символ, указывающий на то, как будет трактоваться вводимый аргумент. Остальные необязательные элементы задают параметры форматирования:

[*]	Запрет занесения в память читаемого поля. Поле сканируется, но его значение не присваивается аргументу из списка.	
[width]	Ширина поля — максимальное число читаемых символов. Реально может быть прочитано меньше символов, если во входном потоке раньше встретится пробельный символ или символ, который не может быть преобразован согласно заданному формату.	
[F N]	Модификаторы, изменяющие размер аргумента по умолчанию:	
	N	ближний указатель (near)
	F	дальний указатель (far)
[h l L]	Модификаторы, изменяющие размер аргумента по умолчанию:	
	h	short int
	l	long int , если type соответствует целому числу, или double , если type соответствует действительному числу
	L	long double

Ниже приведены возможные значения **type**.

Символ	Ожидаемый тип данных	Тип аргумента
Числа		
d	целый	указатель на int (int *arg)
D	целый	указатель на long (long *arg)

Символ	Ожидаемый тип данных	Тип аргумента
e,E	действительный	указатель на float (float *arg)
f	действительный	указатель на float (float *arg)
g,G	действительный	указатель на float (float *arg)
o	восьмеричный целый	указатель на int (int *arg)
O	восьмеричный целый	указатель на long (long *arg)
i	десятичный, восьмеричный или шестнадцатеричный целый	указатель на int (int *arg)
I	десятичный, восьмеричный или шестнадцатеричный целый	указатель на long (long *arg)
u	десятичный целый без знака	указатель на unsigned int (unsigned int *arg)
U	десятичный целый без знака	указатель на unsigned long (unsigned long *arg)
x	шестнадцатеричный целый	указатель на int (int *arg)
X	шестнадцатеричный целый	указатель на int (int *arg)
Символы		
s	строка символов	указатель на массив символов (char arg[])
c	символ	указатель на char (char *arg) или, если задана ширина поля (например, %5c), то на массив символов размером W (char arg[W])
%	символ %	не преобразуется
Указатели		
p	указатель на int (int *arg)	в ячейку памяти, на которую указывает аргумент, заносится количество успешно прочитанных к данному моменту символов
p	шестнадцатеричный формат: YYYY:ZZZ или ZZZZ	указатель на объект (far* или near*)

Примеры использования строки форматирования вы можете найти в разделе 15.5.4 в описании функций ввода.

15.1.4.3 Строка форматирования функций типа **Format**

Описанная ниже строка форматирования используется в функциях **Format**, **FormatBuf**, **FmtStr**, **StrFmt**, **StrLFmt** и др.

Строка содержит обычные символы и спецификаторы формата полей. Обычные символы просто копируются в выходную строку, а спецификаторы определяют форматирование аргументов из заданного списка.

Спецификации формата начинаются с символа **%** и имеют вид:

```
"%" [index ":" ] ["-" ] [width] ["." prec] type
```

Единственно обязательным элементом спецификации является **type** — символ, указывающий на то, как будет трактоваться аргумент. Остальные необязательные элементы задают параметры форматирования:

[index ":"]	Устанавливает текущий индекс массива аргументов в заданное значение index . Индексы начинаются с 0. Например, спецификатор %0: переводит индекс на начало массива и обеспечивает повторное форматирование первого аргумента
["-"]	Обеспечивает выравнивание результата влево с заполнением оставшихся правых позиций поля пробелами. В отсутствие спецификатора ["-"] выравнивание производится вправо
[width]	Устанавливает минимальную ширину поля в результирующей строке. Если результат преобразования короче ширины поля, происходит выравнивание вправо (или влево, если был записан спецификатор ["-"]) с заполнением лишних позиций пробелами
["." prec]	Спецификатор точности, определяющий число выводимых символов (в зависимости от принятого формата). Спецификатор записывается после символа точки (.), чтобы отделить его от предшествующего спецификатора width

Значения спецификаторов **index**, **width** и **prec** могут задаваться в виде целых значений или в виде символа звездочки (*). В последнем случае предполагается, что значение спецификатора задается очередным аргументом из списка.

Ниже приведены возможные значения **type**.

Символ	Тип аргумента	Формат вывода
d	целый	Десятичный формат — строка десятичных цифр. Если используется спецификатор ["." prec] , то он указывает минимальное количество выводимых цифр. Если действительное количество цифр результата меньше указанного спецификатором точности, то происходит выравнивание вправо с заполнением лишних позиций нулями.
e	действительный	Научный формат — строка вида «-d.ddd...E+ddd» . Перед десятичной точкой всегда помещается одна цифра и для отрицательных величин — знак минус. Если используется спецификатор ["." prec] , то он указывает общее количество выводимых цифр, включая цифру перед десятичной точкой (по умолчанию 15 цифр). После символа порядка «E» всегда указывается знак — плюс или минус.
f	действительный	Формат с фиксированной точкой — строка вида «-ddd.ddd...» . Если используется спецификатор ["." prec] , то он указывает количество выводимых цифр после десятичной точки (по умолчанию 2 цифры).

Символ	Тип аргумента	Формат вывода
g	действительный	Универсальный формат — преобразование в научный формат или формат с фиксированной точкой, в зависимости от того, какой из них дает более компактный результат. Если используется спецификатор ["." prec] , то он указывает количество выводимых значащих разрядов (по умолчанию — 15). Начальные нули не печатаются, десятичная точка печатается, если необходимо. Формат с фиксированной точкой используется, если в преобразуемом значении число цифр до десятичной точки меньше заданной точности и если значение не меньше 0.00001. В остальных случаях используется научный формат.
p	действительный	Числовой формат — то же, что формат с фиксированной точкой, но с добавлением разделителей тысяч: «-d,ddd,ddd.ddd...» .
m	действительный	Монетарный формат — число преобразуется в строку, отображающую денежную сумму. Формат контролируется глобальными переменными CurrencyString , CurrencyFormat , NegCurrFormat , ThousandSeparator , DecimalSeparator , CurrencyDecimals , задаваемыми для монетарного формата разделом Currency Format элемента International Контрольной панели Windows. Если используется спецификатор ["." prec] , то он заменяет собой значение глобальной переменной CurrencyDecimals .
p	указатель	Указатель — значение преобразуется в строку вида «XXXX:YYYY» , где XXXX и YYYY — сегмент и смещение, выраженные четырьмя шестнадцатеричными цифрами.
s	символ, строка или тип PChar	Строка символов. Если используется спецификатор ["." prec] , то он задает максимальное число символов. Если строка длиннее указанного числа, она усекается.
x	целый	Шестнадцатеричный формат — строка шестнадцатеричных цифр. Если используется спецификатор ["." prec] , то он указывает минимальное количество цифр результата. Если результат короче, лишние позиции слева заполняются нулями.

Все указанные в приведенной таблице обозначения форматов могут записываться в нижнем или верхнем регистре, что никак не влияет на результат.

Для форматов действительных чисел реально используемые символы десятичной точки и разделителей тысяч определяются глобальными переменными **DecimalSeparator** и **ThousandSeparator**.

Примеры

Примеры влияния формата:

число	%f	%e	%g
110000.	110000,00	1,1000000000000000E+005	110000
-1.1e+08	-110000000,00	-1,1000000000000000E+008	-110000000
0.00011	0,00	1,1000000000000000E-004	0,00011
1.1e-07	0,00	1,1000000000000000E-007	1,1E-7
12.	12,00	1,2000000000000000E+001	12
0.	0,00	0,0000000000000000E+000	0

Примеры влияния точности:

спецификация \ число	1,1E-4	12.	0,00
%.2f	0,00	12,00	0,000
%.3f	0,000	12,000	0,0000
%.4f	0,0001	12,0000	0,0E+000
%.2e	1,1E-007	1,2E+001	0,0E+000
%.3e	1,10E-007	1,20E+001	0,00E+000
%.4e	1,100E-007	1,200E+001	0,000E+000
%.2g	1,1E-7	12	0
%.3g	1,1E-7	12	0
%.4g	1,1E-7	12	0

15.1.4.4 TFloatFormat и TFloatValue — типы форматирования действительных чисел

Типы **TFloatFormat** и **TFloatValue** определяют форматирование действительных чисел в таких функциях, как **FloatToText**, **FloatToStrF**, **FloatToDecimal**, **TextToFloat**.

Синтаксис

```
#include <SysUtils.hpp>
enum TFloatFormat { ffGeneral, ffExponent, ffFixed, ffNumber,
                   ffCurrency };
enum TFloatValue { fvExtended, fvCurrency };
```

Описание

TFloatValue указывает тип преобразуемого числа. Значение **fvExtended** соответствует обычному числу с плавающей запятой, а значение **fvCurrency** — числу типа **Currency**.

Тип **TFloatFormat** определяет коды форматирования чисел с плавающей запятой в функциях **FloatToText**, **FloatToStrF**, **FloatToDecimal**, **TextToFloat**. Возможные значения формата определяют следующие правила форматирования:

ffGeneral	Основной числовой формат. Число преобразуется по формату с фиксированной точкой или научному в зависимости от того, какой из них оказывается короче. Начальные нули удаляются, десятичная точка ставится только при необходимости. Фиксированный формат используется, если число разрядов слева от точки не больше указанной точности Precision и если значение не меньше 0.00001. В противном случае используется научный формат, в котором параметр Digits определяет число разрядов степени — от 0 до 4
ffExponent	Научный формат. Число преобразуется в строку вида «-d.ddd...E+dddd». Общее число цифр, включая одну перед десятичной точкой, задается параметром Precision . После символа «E» всегда следует знак «+» или «-» и до четырех цифр. Параметр Digits определяет минимальное число разрядов степени — от 0 до 4
ffFixed	Формат с фиксированной точкой. Число преобразуется в строку вида «-ddd.ddd...». По крайней мере одна цифра всегда предшествует десятичной точке. Число цифр после десятичной точки задается параметром Digits , который может лежать в пределах от 0 до 18. Если число разрядов слева от десятичной точки больше указанного параметром Precision , то используется научный формат
ffNumber	Числовой формат. Число преобразуется в строку вида «-d,ddd,ddd.ddd...». Данный формат совпадает с ffFixed за исключением наличия в нем разделителей тысяч
ffCurrency	Монетарный формат. Число преобразуется в строку, отображающую денежную сумму. Формат контролируется глобальными переменными CurrencyString , CurrencyFormat , NegCurrFormat , ThousandSeparator , DecimalSeparator , задаваемыми для монетарного формата разделом Currency Format элемента International Контрольной панели Windows. Число цифр после десятичной точки задается параметром Digits , который может лежать в пределах от 0 до 18

Для всех форматов действительные символы, используемые в качестве десятичной точки и разделителя тысяч определяются глобальными переменными **DecimalSeparator** и **ThousandSeparator**.

15.1.4.5 Строка форматирования функций типа **FormatFloat**

Строка форматирования, описанная ниже, применяется в функциях **FormatFloat**, **FloatToTextFmt**, в методе **FormatFloat** класса **AnsiString** и в некоторых других.

В строке используются следующие символы:

- 0** Сохранение позиции для цифры. Если форматируемое число содержит цифру в позиции, в которой в строке форматирования имеется символ «0», то эта цифра копируется в выходную строку. В противном случае в этой позиции в выходной строке содержится «0»
- #** Сохранение позиции для цифры. Если форматируемое число содержит цифру в позиции, в которой в строке форматирования имеется символ «#», то эта цифра копируется в выходную строку. В противном случае в эту позицию в выходной строке ничего не заносится

- Десятичная точка. Первый символ точки «.» в строке форматирования определяет позицию десятичной точки в отформатированном числе. Любые последующие символы «.» в строке игнорируются. Действительный символ, используемый в качестве десятичной точки, определяется глобальной переменной **DecimalSeparator**, установленной в разделе Number Format элемента International программы «Панель управления» Windows
- , Разделитель тысяч. Если строка форматирования содержит один или более символов «,», то в выходной строке будут использованы разделители тысяч. Местоположение символов «,» в строке форматирования безразлично — это просто указание, что надо использовать разделители тысяч. Действительный символ, используемый в качестве разделителя, определяется глобальной переменной **ThousandSeparator**, установленной в разделе Number Format элемента International программы «Панель управления» Windows
- E+, E-, e+, e- Научный формат. Если в строке форматирования встречаются символы «E+», «E-», «e+» или «e-», то при форматировании используется научный формат. Сразу после этих символов может быть расположена группа символов «0» (до четырех символов), которая определяет минимальное число цифр в показателе степени. Если в строке использованы символы «E+» или «e+», то как перед положительной, так и перед отрицательной степенью будет всегда помещаться знак «+» или «-». Если в строке использованы символы «E-» или «e-», то знак будет помещаться только перед отрицательной степенью
- 'xx' / "xx" Символы, заключенные в одинарные или двойные кавычки, выводятся в выходную строку, никак не влияя на форматирование
- ; Символ разделяет разделы строки, связанные с форматированием положительных, отрицательных и нулевых значений

Расположение крайнего левого символа «0» перед десятичной точкой и крайнего правого символа «0» после десятичной точки определяет число цифр, всегда присутствующих в выходной строке.

Форматируемое число всегда округляется до стольких десятичных разрядов, сколько символов «0» и «#» находится справа от десятичной точки. Если строка форматирования не содержит десятичной точки, значение формируемого числа округляется до ближайшего целого.

Если формируемое число имеет больше цифр слева от десятичной точки, чем количество расположенных там в строке форматирования символов «0» и «#», то лишние цифры все равно выводятся в начале числа.

Строка форматирования может содержать от одной до трех секций, разделяемых точкой с запятой. Если задана только одна секция, то она применяется для форматирования любых чисел. Если задано две секции, то первая используется при форматировании положительных чисел и нуля, а вторая — при форматировании отрицательных чисел. Если заданы три секции, то первая относится к положительным числам, вторая — к отрицательным, третья — к нулю.

Если секции отрицательных чисел или нуля пустые (т.е. ничего не написано после соответствующей точки с запятой, то вместо них используется секция положительных чисел.

Если секция положительных чисел пустая или вообще строка форматирования пустая, то используется основной формат чисел с плавающей запятой с 15 значащими разрядами. Этот формат соответствует формату **ffGeneral** типа **TFloatFormat** (см. раздел 15.1.4.4). Этот же формат используется, если число имеет более 18 разрядов до десятичной точки и строка форматирования не содержит указания на применение научного формата.

Примеры

Ниже приведены строки форматирования и соответствующие им выходные строки.

Строка форматирования				
пустая	1234	-1234	0.5	0
0	1234	-1234	1	0
0.00	1234,00	-1234,00	0,50	0,00
###	1234	-1234	,5	
###0.00	1 234,00	-1 234,00	0,50	0,00
###0.00;(###0.00)	1 234,00	(1 234,00)	0,50	0,00
###0.00;;Нуль	1 234,00	-1 234,00	0,50	Нуль
0.000E+00	1,234E+03	-1,234E+03	5,000E-01	0,000E+00
####E-0	1,234E3	-1,234E3	5E-1	0E0

15.1.5 Обработка ошибок времени выполнения, диагностика

Чтобы работать с сообщениями об ошибках времени выполнения, в приложении должна быть включена директива

```
#include <errno.h>
```

15.1.5.1 `_doserrno`, `errno` и `_sys_nerr` — переменные, содержащие коды ошибок

Переменные `_doserrno` и `errno` типа `int` получают положительные значения при возникновении различных ошибок времени выполнения. Значения `_doserrno` и `errno` задаются одновременно, но иногда они могут различаться, поскольку `errno` — переменная, совместимая с UNIX.

Коды ошибок, присваиваемые `errno`, являются одновременно индексами массива `_sys_errlist`, содержащего сообщения об ошибках. Кроме того имеется переменная `_sys_nerr`, которая содержит номер ошибки и используется функцией `perror` (файл `stdio.h`) для вывода в стандартный поток сообщений об ошибках `stderr`. Поэтому доступ к соответствующему сообщению можно получить или как `_sys_errlist[errno]`, или как `_sys_errlist[_sys_nerr]`.

Нормальное значение рассматриваемых переменных — 0. При выполнении различных математических функций, при манипуляциях с файлами и т.п. это значение при возникновении ошибки изменяется и сохраняется таким вплоть до следующего обращения к соответствующей функции. Если это нежелательно, надо программно сбрасывать значения в 0.

15.1.5.2 Коды ошибок

Ниже приводится таблица, содержащая мнемонические константы ошибок, их коды и соответствующие сообщения из массива `_sys_errlist`.

Константа	Код	Сообщение в _sys_errlist
E2BIG	20	Arg list too long
EACCES	5	Permission denied
EAGAIN	42	Resource temporarily unavailable
EBADF	6	Bad file number
EBUSY	44	Resource busy
ECHILD	24	No child process
ECONTR	7	Memory blocks destroyed
ECURDIR	16	Attempt to remove CurDir
EDEADLOCK	36	Locking violation
EDOM	33	Math argument
EEXIST	35	File already exists
EFAULT	14	Unknown error
EFBIG	27	Для UNIX — в MSDOS отсутствует
EINTR	39	Interrupted function call
EINVACC	12	Invalid access code
EINVAL	19	Invalid argument
EINVDAT	13	Invalid data
EINVDRV	15	Invalid drive specified
EINVENV	10	Invalid environment
EINVFMT	11	Invalid format
EINVFNC	1	Invalid function number
EINVMEM	9	Invalid memory block address
EIO	40	Input/output error
EISDIR	46	Для UNIX — в MSDOS отсутствует
EMFILE	4	Too many open files
EMLINK	31	Для UNIX — в MSDOS отсутствует
ENFILE	23	Too many open files
ENMFILE	18	No more files
ENODEV	15	No such device
ENOENT	2	No such file or directory
ENOEXEC	21	Exec format error
ENOFIL	2	File not found
ENOMEM	8	Not enough core
ENOPATH	3	Path not found
ENOSPC	28	No space left on device

Константа	Код	Сообщение в <code>_sys_errlist</code>
ENOTBLK	43	Для UNIX — в MSDOS отсутствует
ENOTDIR	45	Для UNIX — в MSDOS отсутствует
ENOTSAM	17	Not same device
ENOTTY	25	Для UNIX — в MSDOS отсутствует
ENXIO	41	No such device or address
EPERM	37	Operation not permitted
EPIPE	32	Broken pipe
ERANGE	34	Result too large
EROFS	30	Read-only file system
ESPIPE	29	Illegal seek
ESRCH	38	Для UNIX — в MSDOS отсутствует
ETXTBSY	26	Для UNIX — в MSDOS отсутствует
EUCLEAN	47	Для UNIX — в MSDOS отсутствует
EXDEV	22	Cross-device link
EZERO	0	Error 0

Стандартные сообщения можно изменять. Например, оператор

```
strcpy(_sys_errlist[ENOENT], "Нет такого файла или каталога");
```

русифицирует стандартное сообщение «No such file or directory».

Ниже приведена таблица других кодов ошибок — ошибок файлового ввода-вывода, которые возникают, если включена опция I/O checking на странице Pascal окна опций проекта. Эти коды генерируются в C++Builder 5 при создании исключения **EInOutError**.

Код	Ошибка
2	Файл не найден
3	Неправильное имя файла
4	Слишком много открытых файлов
5	Файл не доступен
100	Достигнут конец файла (EOF)
101	Диск переполнен
106	Ошибка ввода

15.1.5.3 EDOM, ERANGE — константы сообщений об ошибках

Символические целочисленные константы **EDOM** и **ERANGE** используются в математических и других функциях для сообщений об ошибках. Узнать, возникла ли соответствующая ошибка при выполнении некоторой функции, можно проверкой переменной **errno**, например:

```
if(errno == EDOM) ...;
```

15.1.5.4 `_matherr` и `_matherrl` — обработчики ошибок

Синтаксис

```
#include <math.h>
int _matherr(struct _exception *e);
int _matherrl(struct _exceptionl *e);
```

Описание

Функция `_matherr` или `_matherrl` (для типов `long double`) вызываются библиотечными математическими функциями при возникновении в них ошибок, связанных с недопустимыми значениями параметров (корень или логарифм отрицательного числа и т.п.). Функции перехватывают только ошибки, выхода за пределы области определения и выхода за диапазон допустимых значений, но не реагируют на исключения при выполнении математических операций, например, при делении на 0. Для перехвата таких событий служит функция `signal`.

Стандартные варианты `_matherr` и `_matherrl` могут быть переопределены пользователем, если он объявит в своем приложении аналогичные функции. Эти функции пользователя должны возвращать ненулевое значение, если они обрабатывали ошибку. В этом случае не возникает стандартного сообщения об ошибке и не изменяется значение переменной `errno`. Если переписанные пользователем варианты `_matherr` и `_matherrl` не обработали данную ошибку, они должны вернуть 0. Тогда будет проведена стандартная обработка ошибки.

В качестве параметра `e` в функции передаются структуры:

```
struct _exception {
    int    type;
    char   *name;
    double arg1, arg2, retval;
};

struct _exceptionl {
    int    type;
    char   *name;
    long double arg1, arg2, retval;
};
```

Элемент структуры `type` определяет тип ошибки. Элемент `name` указывает на строку, содержащую имя функции, в которой произошла ошибка. Элементы `arg1` и `arg2` — это значения аргументов, приведшие к ошибкам (если функция имеет один аргумент, то его значение помещается в `arg1`). Элемент `retval` — возвращаемое по умолчанию значение функции. Пользователь может изменить это значение.

Тип ошибки, хранящийся в элементе `type`, может принимать одно из следующих значений:

DOMAIN	аргумент выходит за пределы области определения; например, <code>log(-1)</code>
SING	аргумент соответствует особой точке функции; например, <code>pow(0, -2)</code>
OVERFLOW	аргумент приводит к значению функции, превышающему <code>DBL_MAX</code> (или <code>LDBL_MAX</code>); например, <code>exp(1000)</code>
UNDERFLOW	аргумент приводит к значению функции, меньшему чем <code>DBL_MIN</code> (или <code>LDBL_MIN</code>); например, <code>exp(-1000)</code>
TLOSS	аргумент приводит к значению функции с полной потерей значащих разрядов; например, <code>sin(10e70)</code>

Фигурирующие в приведенном описании макросы **DBL_MAX**, **DBL_MIN**, **LDBL_MAX** и **LDBL_MIN** определены в файле **float.h**.

Пример

Приведенный ниже пример показывает функцию, обрабатывающую ошибку типа **DOMAIN** функции **sqrt** (корень из отрицательного числа), заменяя результат на корень из положительного числа:

```
int _matherr (struct _exception *a)
{
    if (a->type == DOMAIN)
        if (!strcmp(a->name, "sqrt")) {
            a->retval = sqrt (-(a->arg1));
            return 1;
        }
    return 0;
}
```

15.1.5.5 assert — макрос диагностики

Синтаксис

```
#include <assert.h>
void assert(int test);
```

Описание

Макрос **assert** используется в программах для диагностики. Если при расширении макроса значение параметра **test** ложно (равно нулю), то **assert** выдает в стандартный файл ошибок **stderr** сообщение:

Assertion failed: test, file <имя файла>, line <номер строки>

После этого макрос **assert** производит вызов функции **abort**.

Если в исходном файле перед директивой

```
#include <assert.h>
```

появляется директива препроцессора

```
#define NDEBUG
```

то все последующие макросы **assert** игнорируются. Таким образом вы можете ввести в свое приложение какие-то проверки, необходимые для отладки, а затем в окончательном файле отключить их директивой **NDEBUG**.

15.2 Математические функции

15.2.1 Константы, используемые в математических выражениях

Константа	Описание	Значение
M_1_PI	1 / π	0.318309886183790671538
M_1_SQRTPI	корень из 1 / π	0.564189583547756286948
M_2_PI	2 / π	0.636619772367581343076
M_2_SQRTPI	2 / корень из π	1.12837916709551257390
M_E	число e	2.71828182845904523536

Константа	Описание	Значение
M_LN10	$\ln(10)$ — логарифм натуральный от 10	2.30258509299404568402
M_LN2	$\ln(2)$ — логарифм натуральный от 2	0.693147180559945309417
M_LOG10E	$\log_{10}(e)$ — логарифм десятичный от e	0.434294481903251827651
M_LOG2E	$\log_2(e)$ — логарифм по основанию 2 от e	1.44269504088896340736
M_PI	число π	3.14159265358979323846
M_PI_2	$\pi / 2$	1.57079632679489661923
M_PI_4	$\pi / 4$	0.785398163397448309616
M_SQRT_2	корень из 2, деленный на 2	0.707106781186547524401
M_SQRT2	корень из 2	1.41421356237309504880

15.2.2 Арифметические и алгебраические функции

Функция	Синтаксис	Описание	Файл
_lrotl	<code>unsigned long _lrotl(unsigned long val, int count)</code>	циклический сдвиг val влево на count битов	stdlib.h
_lrotr	<code>unsigned long _lrotr(unsigned long val, int count)</code>	циклический сдвиг val вправо на count битов	stdlib.h
_rotl	<code>unsigned short _rotl(unsigned short value, int count)</code>	циклический сдвиг value влево на count битов	stdlib.h
_rotr	<code>unsigned short _rotr(unsigned short value, int count)</code>	циклический сдвиг value вправо на count битов	stdlib.h
abs	<code>int abs(int x)</code>	абсолютное значение	stdlib.h
cabs	<code>double cabs(struct complex z) struct complex { double x, y; };</code>	модуль комплексного числа z	math.h
cabsl	<code>long double cabsl(struct _complexl z) struct _complex { long double x, y; };</code>	модуль комплексного числа z	math.h
ceil	<code>double ceil(double x)</code>	округление вверх: наименьшее целое, не меньше x	math.h
Ceil	<code>int Ceil(Extended X);</code>	округление вверх: наименьшее целое, не меньше X	Math.hpp

Функция	Синтаксис	Описание	Файл
ceil	<code>long double ceil(long double x)</code>	округление вверх: наименьшее целое, не меньшее x	math.h
div	<code>div_t div(int numer, int denom)</code> <code>typedef struct {</code> <code>int quot; // частное</code> <code>int rem; // остаток</code> <code>} div_t;</code>	целочисленное деле- ние numer / denom	math.h
exp	<code>double exp(double x)</code>	экспонента	math.h
expl	<code>long double expl(long double x)</code>	экспонента	math.h
fabs	<code>double fabs(double x)</code>	абсолютное значение	math.h
fabsl	<code>long double fabsl(long double x)</code>	абсолютное значение	math.h
floor	<code>double floor(double x)</code>	округление вниз: наибольшее целое, не большее x	math.h
Floor	<code>int Floor(Extended X);</code>	округление вниз: наибольшее целое, не большее X	Math.hpp
floorl	<code>long double floorl(long double x)</code>	округление вниз: наибольшее целое, не большее x	math.h
fmod	<code>double fmod(double x, double y)</code>	остаток от деления x / y	math.h
fmodl	<code>long double fmodl(long double x, long double y)</code>	остаток от деления x / y	math.h
frexp	<code>double frexp(double x, int *exponent)</code>	разделяет x на ман- тиссу (возвращает) и степень exponent	math.h
Frexp	<code>void Frexp(Extended X, Extended &Mantissa, int &Exponent)</code>	разделяет X на ман- тиссу Mantissa и сте- пень Exponent	Math.hpp
frexpl	<code>long double frexpl(long double x, int *exponent)</code>	разделяет x на ман- тиссу (возвращает) и степень exponent	math.h
IntPower	<code>Extended IntPower(Extended Base, int Exponent)</code>	возводит Base в це- лую степень Expo- nent	Math.hpp
labs	<code>long labs(long int x)</code>	абсолютное значение	stdlib.h
ldexp	<code>double ldexp(double x, int exp)</code>	$x \cdot 2^{\text{exp}}$	math.h
Ldexp	<code>Extended Ldexp(Extended X, int P)</code>	$x \cdot 2^P$	Math.hpp
ldexpl	<code>long double ldexpl(long double x, int exp)</code>	$x \cdot 2^{\text{exp}}$	math.h

Функция	Синтаксис	Описание	Файл
ldiv	<pre>typedef struct { long int quot; // целое long int rem; // остаток } ldiv_t; ldiv_t ldiv(long int numer, long int denom)</pre>	<p>целочисленное деление:</p> <p>numer / denom;</p> <p>quot — результат</p> <p>rem — остаток</p>	math.h , stdlib.h
LnXP1	Extended LnXP1(Extended X)	натуральный логарифм (X + 1)	Math.hpp
log	double log(double x)	натуральный логарифм	math.h
log10	double log10(double x)	десятичный логарифм	math.h
Log10	Extended Log10(Extended X)	десятичный логарифм	Math.hpp
log10l	long double log10l(long double x)	десятичный логарифм	math.h
Log2	Extended Log2(Extended X)	логарифм по основанию 2	Math.hpp
logl	long double logl(long double x)	натуральный логарифм	math.h
LogN	Extended LogN(Extended Base, Extended X)	логарифм X по основанию Base	Math.hpp
max	max(a, b)	макрос возвращает максимальное значение из a и b любых типов	stdlib.h
min	min(a, b)	макрос возвращает минимальное значение из a и b любых типов	stdlib.h
modf	double modf(double x, double *ipart)	разделяет x на целую часть ipart и возвращаемую дробную часть	math.h
modfl	long double modfl(long double x, long double *ipart)	разделяет x на целую часть ipart и возвращаемую дробную часть	math.h
poly	double poly(double x, int degree, double coeffs[])	полином от x степени degree с коэффициентами coeffs	math.h
Poly	Extended Poly(Extended X, const double * Coefficients, const int Coefficients_Size)	полином от X степени Coefficients_Size с коэффициентами Coefficients	Math.hpp

Функция	Синтаксис	Описание	Файл
polyl	long double polyl(long double x, int degree, long double coeffs[])	полином от x степени degree с коэффициентами coeffs	math.h
pow	double pow(double x, double y)	x^y	math.h
Power	Extended Power(Extended Base, Extended Exponent)	возводит Base в степень Exponent	Math.hpp
powl	long double powl(long double x, long double y)	x^y	math.h
sqrt	double sqrt(double x)	корень квадратный	math.h
sqrtl	long double sqrtl(long double x)	корень квадратный	math.h

Комментарии

При работе с математическими функциями надо иметь в виду, что файлы **math.h** и **Math.hpp** в C++Builder 5 автоматически не подключаются к модулю вашего приложения. Поэтому для использования описанных в этих файлах функций необходимо вручную вводить директивы

```
#include <math.h>
#include <Math.hpp>
```

Функции **exp**, **expl**, **ldexp**, **ldexpl** в случае выхода аргумента за диапазон допустимых значений генерируют ошибку **ERANGE**.

Функции **log**, **log10**, **log10l**, **logl** в случае отрицательного аргумента генерируют ошибку **ERANGE**, а при нулевом аргументе — **EDOM**.

Функции **pow** и **powl** генерируют ошибку **EDOM**, если $x < 0$ и y не является целым числом, а также если $x = 0$ и $y \leq 0$. Возможно также появление ошибки **ERANGE**.

Функции **sqrt** и **sqrtl** генерируют ошибку **EDOM**, если $x < 0$.

Функции файла **Math.hpp** в основном повторяют возможности функций файла **math.h**, но для типа **Extended**.

15.2.3 Тригонометрические функции

Функция	Синтаксис	Описание	Файл
acos	double acos(double x)	арккосинус	math.h
acosl	long double acosl(long double x)	арккосинус	math.h
ArcCos	Extended ArcCos(Extended X)	арккосинус	Math.hpp
ArcCosh	Extended ArcCosh(Extended X)	арккосинус гиперболический	Math.hpp
ArcSin	Extended ArcSin(Extended X)	арксинус	Math.hpp
ArcSinh	Extended ArcSinh(Extended X)	арксинус гиперболический	Math.h
ArcTan2	Extended ArcTan2(Extended Y, Extended X)	арктангенс (Y / X)	Math.hpp

Функция	Синтаксис	Описание	Файл
ArcTanh	Extended ArcTanh(Extended X)	арктангенс гиперболический	Math.hpp
asin	double asin(double x)	арксинус	math.h
asinl	long double asinl(long double x)	арксинус	math.h
atan	double atan(double x)	арктангенс	math.h
atan2	double atan2(double y, double x)	арктангенс y / x	math.h
atan2l	long double atan2l(long double y, long double x)	арктангенс y / x	math.h
atanl	long double atanl(long double x)	арктангенс	math.h
cos	double cos(double x)	косинус	math.h
cosh	double cosh(double x)	косинус гиперболический	math.h
Cosh	Extended Cosh(Extended X)	косинус гиперболический	Math.hpp
coshl	long double coshl(long double x)	косинус гиперболический	math.h
cosl	long double cosl(long double x)	косинус	math.h
Cotan	Extended Cotan(Extended X)	котангенс	Math.hpp
CycleToRad	Extended CycleToRad(Extended Cycles)	вычисляет угол в радианах по его значению в перио- дах Cycles : $2\pi \cdot \text{Cycles}$.	Math.hpp
DegToRad	Extended DegToRad(Extended Degrees)	вычисляет угол в радианах по его значению в градусах Degrees : $\text{Degrees} \cdot \pi / 180$.	Math.hpp
hypot	double hypot(double x, double y)	гипотенуза треугольника с катетами x и y	math.h
Hypot	Extended Hypot(Extended X, Extended Y)	расчет гипотенузы по кате- там X и Y	Math.hpp
hypotl	long double hypotl(long double x, long double y)	гипотенуза треугольника с катетами x и y	math.h
RadToCycle	Extended RadToCycle(Extended Radians)	вычисляет угол в периодах по его значению в радианах Radians : $\text{Radians} / (2\pi)$.	Math.hpp
RadToDeg	Extended RadToDeg(Extended Radians)	вычисляет угол в градусах по его значению в радиа- нах Radians : $\text{Radians} \cdot 180 / \pi$.	Math.hpp
sin	double sin(double x)	синус	math.h

Функция	Синтаксис	Описание	Файл
SinCos	void SinCos(Extended Theta, Extended &Sin, Extended &Cos)	расчет синуса Sin и косинуса Cos угла Theta	Math.hpp
Sinh	Extended Sinh(Extended X)	синус гиперболический	Math.hpp
sinh	double sinh(double x)	синус гиперболический	math.h
sinhl	long double sinhl(long double x)	синус гиперболический	math.h
sinl	long double sinl(long double x)	синус	math.h
Tan	Extended Tan(Extended X)	тангенс	Math.hpp
tan	double tan(double x)	тангенс	math.h
Tanh	Extended Tanh(Extended X)	тангенс гиперболический	Math.hpp
tanh	double tanh(double x)	тангенс гиперболический	math.h
tanh1	long double tanh1(long double x)	тангенс гиперболический	math.h
tanl	long double tanl(long double x)	тангенс	math.h

Комментарии

При работе с тригонометрическими функциями надо иметь в виду, что файлы **math.h** и **Math.hpp** в C++Builder 5 автоматически не подключаются к модулю вашего приложения. Поэтому для использования описанных в этих файлах функций необходимо вручную вводить директивы

```
#include <math.h>
#include <Math.hpp>
```

Во всех тригонометрических функциях угол задается в радианах. Пересчет угла в радианы из значения, заданного в градусах или периодах, позволяют осуществить функции **DegToRad** и **CycleToRad**. Например, оператор

```
double Rad = DegToRad(90);
```

заносит в переменную **Rad** значение угла 90 градусов в радианах. То же самое значение заносит в переменную **Rad** оператор

```
double Rad = CycleToRad(0.25);
```

в котором значение угла задано в периодах (четверть периода). Следующее выражение вычисляет синус 90 градусов:

```
double S = sin(DegToRad(90));
```

Все обратные тригонометрические функции вычисляют главные значения: **acos** и **acosl** — в диапазоне $[0, \pi]$, **asin**, **asinl**, **atan**, **atan2**, **atan2l**, **atanl** — в диапазоне $[-\pi/2, \pi/2]$. Результат возвращается в радианах. Пересчет угла в радианах из значения градусов или долей периода позволяют осуществить функции **RadToDeg** и **RadToCycle**. Например, операторы

```
double A = atan(T);
double A1 = RadToDeg(atan(T));
```

```
double A2 = RadToCycle(atan(T));
```

вычисляют арктангенс **T** в радианах (**A**), в градусах (**A1**) и в долях периода (**A2**).

В функциях **acos**, **acosl**, **asin**, **asini**, если заданный аргумент не попадает в диапазон значений $[-1, +1]$, происходит ошибка выхода за пределы области определения (**EDOM**).

В гиперболических функциях **cosh**, **coshl**, **sinh**, **sinhl**, если заданный аргумент слишком велик, происходит ошибка выхода за диапазон допустимых значений (**ERANGE**).

15.2.4 Генерация псевдослучайных чисел

Функция	Синтаксис / Описание	Файл
_lrand	long _lrand(void) Псевдослучайное целое, диапазон от 0 до $2^{31} - 1$	stdlib.h
rand	int rand(void) Псевдослучайное целое, диапазон от 0 до RAND_MAX	stdlib.h
RandG	Extended RandG(Extended Mean, Extended StdDev) Псевдослучайные числа, распределенные по нормальному закону; Mean — математическое ожидание, StdDev — среднее квадратичное отклонение	Math.hpp
random	int random(int num) Псевдослучайное целое, диапазон от 0 до num - 1	stdlib.h
randomize	void randomize(void) Рандомизация генераторов (кроме RandG) случайной величиной	stdlib.h
Randomize	void Randomize(void) Рандомизация RandG случайной величиной	Math.hpp
srand	void srand(unsigned seed) Рандомизация генераторов (кроме RandG) числом seed	stdlib.h

Комментарии

Функция **rand** возвращает целые псевдослучайные числа, равномерно распределенные в диапазоне от 0 до **RAND_MAX** ($0x7FFFU - 32767$). Длина отрезка аperiodичности псевдослучайных чисел $2^{32} = 4\,294\,967\,296$. Число используемых случайных чисел не должно превышать эту величину. Если вам все-таки требуется больше чисел, то вы должны при приближении к границе отрезка аperiodичности (а лучше задолго до нее) обновить последовательность чисел с помощью функций **randomize** или **srand**.

Если желательно генерировать случайные числа, лежащие в диапазоне от 0 до некоторого значения **N**, то это легко делать операцией вычисления остатка **%**. Например, выражение

```
rand() % 101
```

возвращает числа в диапазоне от 0 до 100, а выражение

```
(rand() % 201) - 100
```

возвращает числа в диапазоне от -100 до 100.

Функцию **rand** можно использовать и для генерации действительных случайных чисел. Например, выражение

```
10. * rand() / RAND_MAX
```

генерирует псевдослучайные действительные числа, распределенные в диапазоне от 0 до 10.

Функция **_lrand** работает аналогично функции **rand**, но имеет отрезок апериодичности 2^{64} и диапазон от 0 до $2^{31} - 1$.

Функция **random** отличается от предыдущих тем, что имеет параметр **num**, определяющий верхнюю границу диапазона генерируемых чисел. Поэтому, если надо, например, генерировать числа в диапазоне от 0 до 100, это можно сделать выражением

```
random(101);
```

не прибегая, как для предыдущих функций, к операции **%**.

Функция **RandG** генерирует квазислучайные действительные числа, распределенные по нормальному закону (закону Гаусса) с математическим ожиданием **Mean** и средним квадратичным отклонением **StdDev**. При работе с этой функцией надо иметь в виду, что файл **Math.hpp** в C++Builder 5 автоматически не подключается к модулю приложения. Поэтому в модуль необходимо вручную вводить директиву

```
#include <Math.hpp>
```

Поскольку генерируемые рассматриваемыми функциями числа являются псевдослучайными, то при каждом новом запуске вашего приложения будет вырабатываться одна и та же последовательность чисел. Если это недопустимо, надо рандомизировать генератор чисел, т.е. задавать ему каждый раз новое случайное исходное число. Рандомизацию всех генераторов, кроме **RandG**, осуществляет функция **randomize**. Достаточно вставить где-то в текст программы (например, в событие **OnCreate** формы) оператор

```
randomize();
```

чтобы при каждом запуске приложения генерировалась новая последовательность чисел.

Функция **srand** отличается от **randomize** тем, что задает в качестве начального не случайное число, а значение своего параметра **seed**.

Рандомизацию генератора **RandG** осуществляет функция **Randomize**, аналогичная **randomize**. Задание конкретного начального числа для этого генератора можно осуществить, задавая значение целой переменной **RandSeed**, определенной в файле **System.hpp**.

15.2.5 Функции обработки статистических данных

Приведенные ниже функции обрабатывают данные, хранящиеся в массиве **Data**, в котором максимальное значение индекса равно **Data_Size**.

Функция	Синтаксис / Описание	Файл
MaxIntValue	int MaxIntValue(const int * Data, const int Data_Size) Максимальное значение	Math.hpp
MaxValue	double MaxValue(const double * Data, const int Data_Size) Максимальное значение	Math.hpp
Mean	Extended Mean(const double * Data, const int Data_Size) Среднее значение (математическое ожидание)	Math.hpp

Функция	Синтаксис / Описание	Файл
MeanAndStdDev	<pre>void MeanAndStdDev(const double * Data, const int Data_Size, Extended &Mean, Extended &StdDev)</pre> <p>Среднее значение Mean и среднее квадратичное отклонение StdDev</p>	Math.hpp
MinIntValue	<pre>int MinIntValue(const int * Data, const int Data_Size)</pre> <p>Минимальное значение</p>	Math.hpp
MinValue	<pre>double MinValue(const double * Data, const int Data_Size)</pre> <p>Минимальное значение</p>	Math.hpp
MomentSkewKurtosis	<pre>void MomentSkewKurtosis(const double * Data, const int Data_Size, Extended &M1, Extended &M2, Extended &M3, Extended &M4, Extended &Skew, Extended &Kurtosis)</pre> <p>Первые четыре момента M1, M2, M3, M4, коэффициент асимметрии Skew, эксцесс Kurtosis</p>	Math.hpp
Norm	<pre>Extended Norm(const double * Data, const int Data_Size)</pre> <p>Эвклидова норма: корень из суммы квадратов</p>	Math.hpp
PopnStdDev	<pre>Extended PopnStdDev(const double * Data, const int Data_Size)</pre> <p>Смещенная оценка среднего квадратичного отклонения</p>	Math.hpp
PopnVariance	<pre>Extended PopnVariance(const double * Data, const int Data_Size)</pre> <p>Смещенная оценка дисперсии (см. Variance)</p>	Math.hpp
StdDev	<pre>Extended StdDev(const double * Data, const int Data_Size)</pre> <p>Несмещенная оценка среднего квадратичного отклонения</p>	Math.hpp
Sum	<pre>Extended Sum(const double * Data, const int Data_Size)</pre> <p>Сумма значений</p>	Math.hpp
SumInt	<pre>int SumInt(const int * Data, const int Data_Size)</pre> <p>Сумма значений</p>	Math.hpp
SumOfSquares	<pre>Extended SumOfSquares(const double * Data, const int Data_Size)</pre> <p>Сумма квадратов значений</p>	Math.hpp

Функция	Синтаксис / Описание	Файл
SumsAndSquares	void SumsAndSquares(const double * Data, const int Data_Size, Extended &Sum, Extended &SumOfSquares) Сумма Sum и сумма квадратов значений SumOfSquares	Math.hpp
TotalVariance	Extended TotalVariance(const double * Data, const int Data_Size) Сумма квадратов отклонений от среднего значения	Math.hpp
Variance	Extended Variance(const double * Data, const int Data_Size) Несмещенная оценка дисперсии (см. PopnVariance)	Math.hpp

Комментарии

Файл **Math.hpp** в C++Builder 5 автоматически не подключается к модулю приложения. Поэтому в модуль необходимо вручную вводить директиву

```
#include <Math.hpp>
```

Функция **MeanAndStdDev** рассчитывает среднее значение (математическое ожидание) и среднее квадратичное отклонение за один проход. Поэтому расчет выполняется вдвое быстрее, чем при поочередном применении функций **Mean** и **StdDev**. Точность вычислений может быть несколько пониженной при очень больших значениях математического ожидания ($> 10^7$) или очень малых дисперсиях.

Значение среднего квадратичного отклонения, возвращаемое функциями **MeanAndStdDev** и **StdDev** — это несмещенная оценка. Она несколько отличается от смещенной оценки, возвращаемой функцией **PopnStdDev**, поскольку при несмещенной оценке сумма квадратов отклонений делится на $(n - 1)$, а при смещенной — на n . По той же причине разнятся значения дисперсий, возвращаемые функциями **Variance** и **PopnVariance**.

Тест для проверки функций статистической обработки данных может иметь, например, следующий вид:

```
double A[1001];
long double M, StdD, StdD2, M1, M2, M3, M4, Skew, Kurtosis;
// заполнение массива нормально распределенными числами
for(int i = 0; i < 1001; i++)
    A[i] = RandG(20, 4);

double y = Variance(A, 1000);
MeanAndStdDev(A, 1000, M, StdD);
StdD2 = PopnStdDev(A, 1000);
MomentSkewKurtosis(A, 1000, M1, M2, M3, M4, Skew, Kurtosis);
...
```

15.2.6 Вспомогательные функции

Функция	Синтаксис / Описание	Файл
_clear87	unsigned int _clear87 (void) Очищает слово статуса плавающей запятой	float.h

Функция	Синтаксис / Описание	Файл
_control87	unsigned int _control87(unsigned int newcw, unsigned int mask) Манипулирует словом контроля плавающей запятой	float.h
_fpreset	void _fpreset(void) Повторно инициализирует пакет математики с плавающей запятой	float.h
_status87	unsigned int _status87(void) Возвращает статус плавающей точки	float.h

15.3 Преобразование типов данных

15.3.1 Функции взаимного преобразования чисел и строк

15.3.1.1 Функции взаимного преобразования чисел и строк типа char *

Функция	Синтаксис / Преобразует	Файл
_atoi64	__int64 _atoi64(const char *s) Строку s в целое	stdlib.h
_atold	long double _atold(const char *s) Строку s в число с плавающей запятой	math.h
_i64toa	char *_i64toa(__int64 value, char *strP, int radix) Целое value в строку; radix — основание (от 2 до 36)	stdlib.h
_itow	wchar_t *_itow(int value, wchar_t *string, int radix) Целое value в строку string по основанию radix	stdlib.h
_ltoa	char *_ltoa(long value, char *string, int radix) Целое value в строку; radix — основание (от 2 до 36)	stdlib.h
_strtold	long double _strtold(const char *s, char **endptr) Строки s в действительное число	stdlib.h
_ui64toa	char *_ui64toa(unsigned __int64 value, char *strP, int radix) Целое value в строку; radix — основание (от 2 до 36)	stdlib.h
_ultow	wchar_t *_ultow(unsigned long value, wchar_t *string, int radix) Целое value в строку string по основанию radix	stdlib.h
_westold	long double _westold(const wchar_t *s, wchar_t **endptr) Строку s в действительное число	stdlib.h
_wtof	double _wtof(const wchar_t *s) Строку s в число с плавающей запятой	math.h
_wtoi	int _wtoi(const wchar_t *s) Строку s в целое	stdlib.h

Функция	Синтаксис / Преобразует	Файл
_wtoi64	__int64 _wtoi64(const wchar_t *s) Строку <i>s</i> в целое	stdlib.h
_wtol	long _wtol(const wchar_t *s) Строку <i>s</i> в целое	stdlib.h
_wtdold	long double _wtdold(const wchar_t *s) Строку <i>s</i> в число с плавающей запятой	math.h
atof	double atof(const char *s) Строку <i>s</i> в число с плавающей запятой	stdlib.h, math.h
atoi	int atoi(const char *s) Строку <i>s</i> в целое	stdlib.h
atol	long atol(const char *s) Строку <i>s</i> в целое	stdlib.h
ecvt	char *ecvt(double value, int ndig, int *dec, int *sign) Число с плавающей запятой <i>value</i> в строку с числом цифр <i>ndig</i> ; <i>dec</i> сохраняет позицию десятичной точки, <i>sign</i> — знак	stdlib.h
fcvt	char *fcvt(double value, int ndig, int *dec, int *sign) Число с плавающей запятой <i>value</i> в строку с числом цифр <i>ndig</i> ; <i>dec</i> сохраняет позицию десятичной точки, <i>sign</i> — знак	stdlib.h
gcvt	char *gcvt(double value, int ndec, char *buf) <i>value</i> в строку <i>buf</i> с числом цифр <i>ndec</i>	stdlib.h
itoa	char *itoa(int value, char *string, int radix) Целое <i>value</i> в строку <i>string</i> по основанию <i>radix</i>	stdlib.h
strtod	double strtod(const char *s, char **endptr) Строку <i>s</i> в действительное число	stdlib.h
strtol	long strtol(const char *s, char **endptr, int radix) Строку <i>s</i> в длинное целое	stdlib.h
strtoul	unsigned long strtoul(const char *s, char **endptr, int radix) Строку <i>s</i> в unsigned long по основанию <i>radix</i>	stdlib.h
ultoa	char *ultoa(unsigned long value, char *string, int radix) Целое <i>value</i> в строку <i>string</i> по основанию <i>radix</i>	stdlib.h
westod	double westod(const wchar_t *s, wchar_t **endptr) Строку <i>s</i> в действительное число	stdlib.h
westol	long westol(const wchar_t *s, wchar_t **endptr, int radix) Строку <i>s</i> в длинное целое	stdlib.h
westoul	unsigned long westoul(const wchar_t *s, wchar_t **endptr, int radix) Строку <i>s</i> в unsigned long по основанию <i>radix</i>	stdlib.h

Комментарии

Функции преобразования строки в число требуют, чтобы строка была записана в формате чисел соответствующего типа. Преобразование прерывается, когда функция встречает первый символ, не соответствующий требуемому формату. Если формат вообще не соответствует ожидаемому, функции возвращают 0.

Функции **atof** и **strtod** распознают кроме соответствующих цифровых последовательностей тексты «+INF» и «-INF», которыми обозначаются плюс и минус бесконечности, а также тексты «+NAN» и «-NAN», обозначающие «не цифровая величина».

В функциях **strtod**, **strtol**, **_strtold**, **strtoul**, **wctod**, **wctol**, **wctoul** параметр **endptr** может задаваться равным **NULL**. Например, оператор:

```
double y = strtod(Edit1->Text.c_str(), NULL);
```

преобразует текст, введенный пользователем в окне редактирования **Edit1**, в значение **y**. Если же задать параметр **endptr**:

```
char *endptr;  
double y = strtod(Edit1->Text.c_str(), &endptr);
```

то величина ***endptr** будет равна тому символу, на котором остановилось преобразование строки. Этот параметр можно использовать для проверки правильности преобразуемой строки.

Если при преобразовании наступает переполнение, то функции возвращают положительные или отрицательные значения **HUGE_VAL** (для типа **double**) или **LHUGE_VAL** (для типа **long double**).

Рассмотренные функции преобразования можно использовать и для типа строк **AnsiString** (см. раздел 15.4.2.3). Но при этом эти строки надо переводить в тип **char *** с помощью метода **c_str()**, как показано в двух предыдущих примерах.

15.3.1.2 Функции взаимного преобразования чисел и строк, описанные в файле SysUtils.hpp

Функция	Синтаксис / Преобразует
CurrToStr	System::AnsiString CurrToStr(System::Currency Value) Число Value типа Currency в строку
CurrToStrF	System::AnsiString CurrToStrF(System::Currency Value, TFloatFormat Format, int Digits) Число типа Currency в строку с помощью формата типа TFloatFormat (см. раздел 15.1.4.4)
FloatToDecimal	void FloatToDecimal(TFloatRec &Result, const void *Value, TFloatValue ValueType, int Precision, int Decimals) Число Value типа ValueType (см. TFloatValue в разделе 15.1.4.4) в структуру TFloatRec
FloatToStr	System::AnsiString FloatToStr(Extended Value) Число Value в строку
FloatToStrF	System::AnsiString FloatToStrF(Extended Value, TFloatFormat Format, int Precision, int Digits) Число Value в строку с помощью формата типа TFloatFormat (см. раздел 15.1.4.4)

Функция	Синтаксис / Преобразует
FloatToText	int FloatToText(char * Buffer, const void *Value, TFloatValue ValueType, TFloatFormat Format, int Precision, int Digits) Число Value типа ValueType в строку Buffer с помощью формата типа TFloatFormat (см. раздел 15.1.4.4)
FloatToTextFmt	int FloatToTextFmt(char * Buffer, const void *Value, TFloatValue ValueType, char * Format) Число Value типа ValueType (см. TFloatValue в разделе 15.1.4.4) в строку Buffer с помощью формата FormatFloat (см. раздел 15.1.4.5)
FmtStr	void FmtStr(System::AnsiString &Result, const System::AnsiString Format, const System::TVarRec * Args, const int Args_Size) Аргументы из открытого массива Args размера Args_Size - 1 в строку Result по формату Format (см. раздел 15.1.4.3)
Format	System::AnsiString Format(const System::AnsiString Format, const System::TVarRec* Args, const int Args_Size) Аргументы из открытого массива Args размера Args_Size - 1 в строку по формату Format (см. раздел 15.1.4.3)
FormatBuf	Cardinal FormatBuf(void *Buffer, Cardinal BufLen, const void *Format, Cardinal FmtLen, const System::TVarRec * Args, const int Args_Size) Аргументы из открытого массива Args размера Args_Size - 1 в строку Buffer длины BufLen по формату Format (см. раздел 15.1.4.3) длины FmtLen
FormatCurr	System::AnsiString FormatCurr(const System::AnsiString Format, System::Currency Value) Число типа Currency в строку с помощью формата функции FormatFloat (см. раздел 15.1.4.5)
FormatFloat	System::AnsiString FormatFloat(const System::AnsiString Format, Extended Value) Число Value в возвращаемую строку с помощью формата типа FormatFloat (см. раздел 15.1.4.5)
GetFormatSettings	void GetFormatSettings(void) Устанавливает значения по умолчанию всех глобальных переменных, определяющих форматы дат и чисел
IntToHex	System::AnsiString IntToHex(int Value, int Digits) Целое Value в строку с минимум Digits шестнадцатеричных цифр
IntToStr	System::AnsiString IntToStr(int Value) Целое Value в строку

Функция	Синтаксис / Преобразует
StrFmt	char * StrFmt(char * Buffer, char * Format, const System::TVarRec * Args, const int Args_Size) Аргументы из открытого массива Args размера Args_Size - 1 в строку Buffer по формату Format (см. раздел 15.1.4.3)
StrLFmt	char * StrLFmt(char * Buffer, Cardinal MaxLen, char * Format, const System::TVarRec* Args, const int Args_Size) Аргументы из открытого массива Args размера Args_Size - 1 в строку Buffer размера MaxLen по формату Format (см. раздел 15.1.4.3)
StrToCurr	System::Currency StrToCurr(const System::AnsiString S) Строку S в число типа Currency
StrToFloat	Extended StrToFloat(const System::AnsiString S) Строку S в число
StrToInt	int StrToInt(const System::AnsiString S) Строку S в целое
StrToIntDef	int StrToIntDef(const System::AnsiString S, int Default) Строку S в целое, при ошибке — значение Default по умолчанию
TextToFloat	bool TextToFloat(char * Buffer, void *Value, TFloatValue ValueType) Строку Buffer в число Value типа ValueType (см. TFloatValue в разделе 15.1.4.4)

Комментарии

Многие функции взаимного преобразования чисел и строк, объявленные в файле **SysUtils.hpp**, используют для указания типа числа переменную **ValueType**, которая может принимать значение **fvExtended** — число с плавающей запятой типа **Extended**, или значение **fvCurrency** — число типа **Currency**. Многие функции используют для форматирования строку типа **TFloatFormat**, подробно описанную в разделе 15.1.4.4, или формат функции **FormatFloat**, описанный в разделе 15.1.4.5, или строку форматирования функции, **Format**, описанную в разделе 15.1.4.3.

Ряд функций получает список форматируемых значений из открытого массива аргументов **Args** размера **Args_Size - 1**. В качестве **Args_Size** в них задается последний индекс массива **Args** типа **TVarRec**. В этих функциях используется строка форматирования, описанная в разделе 15.1.4.3. Приведем пример использования одной из таких функций — функции **Format**:

```
AnsiString s;
TVarRec Args[3] = {11, -1.1e+08, 0.00011};
s = Format("%d %g %f %0:10d %10g %10.5f", Args, 2);
```

В этом примере объявлена переменная **s**, в которую производится запись результатов форматирования, и массив **Args**, в который занесены форматируемые числа. Обратите внимание на то, что размер массива равен 3, а в функцию **Format** передается в качестве размера числа 2 — максимальное значение индекса, на 1 меньшее размера. Если бы нужно было форматировать не все три, а, например, только два первых числа массива, то в качестве размера можно было бы передать 1.

Строка форматирования в этом примере сначала заносит в строку **s** значения элементов массива по форматам **%d**, **%g**, **%f**, оставляя между ними пробелы (пробелы между спецификациями в строке форматирования переносятся в строку результата). Затем спецификация **%0:10d** сбрасывает индекс массива на 0, приводя к повторному форматированию элементов массива. Повторно они форматируются с заданной шириной поля 10, а последнее число еще и с заданной точностью 5.

Для ссылки на массив аргументов можно было бы воспользоваться макросом **EXISTINGARRAY** (см. раздел 15.7.4):

```
s = Format("%d %g %f %0:10d %10g %10.5f", EXISTINGARRAY(Args));
```

Можно было бы и не создавать заранее массива аргументов, а сформировать его непосредственно в вызове функции **Format** с помощью макроса **OPENARRAY** (см. раздел 15.7.4):

```
s = Format("%d %g %f %0:10d %10g %10.5f",  
          OPENARRAY(TVarRec, ( 11, -1.1e+08, 0.00011)));
```

При ошибках преобразования рассматриваемые в данном разделе функции генерируют исключение **EConvertError**. Это правило не затрагивает функцию **StrToIntDef**, которая в случае ошибки заносит в результат указанное в ней значение по умолчанию.

Функция **FloatToDecimal** преобразует число с плавающей запятой типа **Extended** или **Currency** в десятичное представление, которое может в дальнейшем подвергаться дополнительному форматированию. Для значения типа **Extended** параметр **Precision** указывает число значащих цифр от 1 до 18. Для значения типа **Currency** параметр **Precision** игнорируется, а точность предполагается равной 19 разрядам.

Параметр **Decimals** указывает максимально требуемое число цифр слева от десятичной точки. Таким образом, параметры **Precision** и **Decimals** совместно определяют способ округления результата. Чтобы результат всегда имел заданное количество значащих цифр независимо от значения числа, можно указать **Decimals** равным 9999.

Результат преобразования заносится в структуру типа **TFloatRec**, имеющую поля:

Exponent	Хранит количество значащих цифр до десятичной точки. Если число меньше 1, то поле Exponent содержит отрицательное число, модуль которого равен номеру первого значащего разряда после десятичной точки. Если значение равно NAN (не число), Exponent равняется -32768. Если значение INF или -INF (плюс или минус бесконечность), то Exponent = 32767
Negative	При отрицательном числе — true , при положительном или нуле — false
Digits	Строка с нулевым символом в конце, содержащая до 18 (для Extended) или 19 (для Currency) значащих цифр. Десятичная точка не хранится. Завершающие нули удаляются. Если число рано нулю, NAN или INF , Digits содержит только нулевой символ

Ниже приведен пример использования функции **FloatToDecimal**:

```
struct TFloatRec Result;  
Extended Value = ...;  
FloatToDecimal(Result, &Value, fvExtended, 18, 9999);
```

При различных значениях **Value** получаются результаты:

Value	Exponent	Negative	Digits
123.4567890123456789	3	false	123456789012345681
1234567890123456789	19	false	123456789012345679
-0.001234567890123456789	-2	true	123456789012345671

15.3.2 Функции преобразования дат и времени

Функция	Синтаксис / Описание	Файл
asctime	char *asctime(const struct tm *tblock) Переводит структуру типа tm в строку	time.h
ctime	char *ctime(const time_t *time) Переводит время time , полученное функцией time , в строку	time.h
Date	System::TDateTime Date(void) Возвращает текущую дату	SysUtils.hpp
DateTimeToFileDate	int DateTimeToFileDate(System::TDateTime DateTime) Переводит DateTime в формат DOS	SysUtils.hpp
DateTimeToStr	System::AnsiString DateTimeToStr(System::TDateTime DateTime) Преобразует DateTime в строку	SysUtils.hpp
DateTimeToString	void DateTimeToString(System::AnsiString &Result, const System::AnsiString Format, System::TDateTime DateTime) Преобразует DateTime в строку Result по формату Format	SysUtils.hpp
DateTimeToSystemTime	void DateTimeToSystemTime(System::TDateTime DateTime, _SYSTEMTIME &SystemTime) Преобразует DateTime в формат TSystemTime , используемый в API Windows	SysUtils.hpp
DateTimeToTimeStamp	TTimeStamp DateTimeToTimeStamp(System::TDateTime DateTime) Преобразует DateTime в TTimeStamp	SysUtils.hpp
DateToStr	System::AnsiString DateToStr(System::TDateTime Date) Преобразует дату Date в строку	SysUtils.hpp
DayOfWeek	int DayOfWeek(System::TDateTime Date) Извлекает из даты Date день недели (от 1 до 7, 1 — воскресенье)	SysUtils.hpp

Функция	Синтаксис / Описание	Файл
DecodeTime	void DecodeTime(System::TDateTime Time, Word &Hour, Word &Min, Word &Sec, Word &MSec) Разбивает Time на часы Hour , минуты Min , секунды Sec , миллисекунды MSec	SysUtils.hpp
EncodeDate	TDateTime EncodeDate(Word Year, Word Month, Word Day) Преобразует год Year , месяц Month и день Day в TDateTime	SysUtils.hpp
EncodeTime	TDateTime EncodeTime(Word Hour, Word Min, Word Sec, Word MSec) Преобразует часы Hour , минуты Min , секунды Sec и миллисекунды MSec в TDateTime	SysUtils.hpp
FormatDateTime	System::AnsiString FormatDateTime(const System::AnsiString Format, System::TDateTime DateTime) Преобразует DateTime в строку по формату Format	SysUtils.hpp
getdate	void getdate(struct date *datep) Заносит в datep текущую дату	dos.h
gettime	void gettime(struct time *timep) Заносит в timep текущее время	dos.h
gmtime	struct tm *gmtime(const time_t *timer) Переводит время timer , полученное функцией time , в структуру типа tm	time.h
IncMonth	System::TDateTime IncMonth(const System::TDateTime Date, int NumberOfMonths) Возвращает дату Date , измененную на NumberOfMonths месяцев	SysUtils.hpp
IsLeapYear	bool IsLeapYear(Word Year) Возвращает true , если год Year високосный	SysUtils.hpp
localtime	struct tm *localtime(const time_t *timer) Переводит время timer , полученное функцией time , в структуру типа tm с поправкой на локальное время	time.h
mktime	time_t mktime(struct tm *t) Переводит время из структуры типа tm в формат time_t	time.h
MSecsToTimeStamp	TTimeStamp MSecsToTimeStamp(System::Comp MSecs) Преобразует миллисекунды MSecs в TTimeStamp	SysUtils.hpp

Функция	Синтаксис / Описание	Файл
Now	System::TDateTime Now(void) Возвращает текущую дату и время	SysUtils.hpp
setdate	void setdate(struct date *datep) Задаёт дату datep как системную (если пользователю разрешен доступ)	dos.h
settime	void settime(struct time *timep) Задаёт время timep как системное	dos.h
stime	int stime(time_t *tp) Задаёт системную дату и время из tp	time.h
StrToDate	System::TDateTime StrToDate(const System::AnsiString S) Преобразует строку S в дату TDateTime	SysUtils.hpp
StrToDateTime	System::TDateTime StrToDateTime(const System::AnsiString S) Преобразует строку S в дату и время TDateTime	SysUtils.hpp
StrToTime	System::TDateTime StrToTime(const System::AnsiString S) Преобразует строку S во время TDateTime	SysUtils.hpp
SystemTimeTo-DateTime	System::TDateTime SystemTimeToDateTime(const _SYSTEMTIME &SystemTime) Преобразует формат TSystemTime , используемый в API Windows, в TDateTime	SysUtils.hpp
time	time_t time(time_t *timer) Возвращает текущее время и заносит его в timer (если timer не NULL)	time.h
Time	System::TDateTime Time(void) Возвращает текущее время	SysUtils.hpp
TimeStampTo-DateTime	System::TDateTime TimeStampToDateTime(const TTimeStamp &TimeStamp) Преобразует структуру типа TTimeStamp в TDateTime	SysUtils.hpp
TimeStampToM-Secs	System::Comp TimeStampToMsecs(const TTimeStamp &TimeStamp) Возвращает 64-разрядное значение числа миллисекунд	SysUtils.hpp
TimeToStr	System::AnsiString TimeToStr(System::TDateTime Time) Преобразует время в строку	SysUtils.hpp

Комментарии

Функции рассмотренного вида используют тип **TDateTime**, представляющий собой число с плавающей запятой, целая часть которого содержит число дней, от-

считанное от 0 часов 12/30/1899, а дробная часть равна части 24-часового дня, отсчитанная от 12 часов дня. Т.е. дробная часть характеризует время и не относится к дате.

Например:

значение TDateTime	дата, время
0	12/30/1899, 00 часов
2.75	1/1/1900, 18 часов
-1.25	12/29/1899, 6 часов утра
35065	1/1/1996, 00 часов

Некоторые функции используют также тип **TTimeStamp**. Это структура вида:

```
struct TTimeStamp
{
    int Time;
    int Date;
};
```

Поле **Date** содержит число дней с начала календаря (т.е. 1 января 1 года отображается как 1). Обратите внимание, что значения дат в типах **TDateTime** и **TTimeStamp** разные. Поле **Time** содержит время в миллисекундах, прошедшее с 0 часов текущего дня.

TDateTime обеспечивает более компактное представление дат и времени. Так что если вам не нужна точность до миллисекунд, лучше пользоваться типом **TDateTime**.

Преобразование структуры типа **TTimeStamp** в **TDateTime** осуществляется функцией **TimeStampToDateTime**, обратное преобразование — функцией **DateTimeToTimeStamp**. Функция **TimeStampToMSecs** возвращает в виде 64-разрядного числа с плавающей запятой общее число миллисекунд, составленное из полей **Date** и **Time**.

Имеется еще один формат представления дат и времени, принятый в DOS. Этот формат используется в таких функциях, как **FileAge**, **FileGetDate**, **FileSetDate**, в поле **Time** структуры типа **TSearchRec**, применяемой в функциях **FindFirst** и **FindNext**. Перевод в этот формат значения типа **TDateTime** осуществляется функцией **DateTimeToFileDate**.

Наконец, имеется еще системный формат — **TSystemTime**, определенный как тип **_SYSTEMTIME**. Он может требоваться при вызове функций API Windows. Преобразование **TDateTime** в этот формат осуществляется функцией **DateTimeToSystemTime**, а обратное преобразование осуществляется функцией **SystemTimeToDateTime**.

В функциях **DateTimeToString** и **FormatDateTime** используется строка форматирования дат, которая может содержать следующие спецификаторы:

- c** Печать даты по формату, заданному глобальной переменной **ShortDateFormat**, а затем печать времени в формате, заданном глобальной переменной **LongTimeFormat**. Если дробная часть **DateTime** равна 0, время не печатается
- d** Печать дня без начального нуля (1 - 31)
- dd** Печать дня с начальным нулем (01 - 31)
- ddd** Печать дня в виде аббревиатуры (пн-вс) с использованием глобальной переменной **ShortDayNames**

dddd	Печать дня в виде его полного названия (понедельник - воскресенье) с использованием глобальной переменной LongDayNames
dddddd	Печать даты по формату, заданному глобальной переменной ShortDateFormat
dddddd	Печать даты по формату, заданному глобальной переменной LongDateFormat
m	Печать месяца без начального нуля (1-12). Но если спецификатор m следует за спецификатором h или hh , то он означает печать минут
mm	Печать месяца с начальным нулем (01-12). Но если спецификатор mm следует за спецификатором h или hh , то он означает печать минут
mmm	Печать месяца в виде аббревиатуры (янв-дек) с использованием глобальной переменной ShortMonthNames
mmmm	Печать полного названия месяца (Январь-Декабрь) с использованием глобальной переменной LongMonthNames
yy	Печать двух последних цифр года (00-99)
yyyy	Полная печать года (0000-9999)
h	Печать часа без начального нуля (0-23)
hh	Печать часа с начальным нулем (00-23)
n	Печать минут без начального нуля (0-59)
nn	Печать минут с начальным нулем (00-59)
s	Печать секунд без начального нуля (0-59)
ss	Печать секунд с начальным нулем (00-59)
t	Печать времени по формату глобальной переменной ShortTimeFormat
tt	Печать времени по формату глобальной переменной LongTimeFormat
am/pm	Использование 12-часовой шкалы и символов « am » или « pm ». Регистр символов совпадает с регистром, в котором записан спецификатор
a/p	Использование 12-часовой шкалы и символов « a » или « p ». Регистр символов совпадает с регистром, в котором записан спецификатор
ampm	Использование 12-часовой шкалы и символов, записанных в глобальные переменные TimeAMString и TimePMString
/	Печать разделителя дат, заданного глобальной переменной DateSeparator
:	Печать разделителя времени, заданного глобальной переменной TimeSeparator
'xx'/'xx'	Символы, заключенные в кавычки, а также любые символы, не совпадающие со спецификаторами, просто переносятся в выходную строку

Примеры форматирования:

формат	результат
"c"	14.04.99 8:34:14
"d/m/yy h:n"	14.4.99 8:34
"Дата: d mmmm ууу г., день — dddd"	Дата: 14 Апрель 1999 г., день — среда

Более простой, но и менее гибкий способ перевода дат и времени в строку, дают функции **DateToStr**, **TimeToStr**, **DateTimeToStr**, использующие установки глобальных переменных.

Несколько функций — **StrToDate**, **StrToDateTime**, **StrToTime** осуществляют обратное преобразование — строки в дату, время или дату и время **TDateTime**. Функция **StrToDate** преобразует строку даты. Строка должна содержать две или три цифры, разделенных символом, указанным в глобальной переменной **DateSeparator**. Последовательность указания дня, месяца и года определяется глобальной переменной **ShortDateFormat**. Возможные варианты: **m/d/y**, **d/m/y** или **y/m/d**. Если указаны только два числа, они интерпретируются как дата (**m/d** или **d/m**) текущего года. Если год указан не более чем двумя цифрами (от 0 до 99), предполагается год текущего столетия.

Функция **StrToTime** преобразует строку времени, которая должна содержать две или три цифры, разделенные символом, указанным в глобальной переменной **TimeSeparator**: **hh:mm:ss**. Секунды могут не указываться. Функция **StrToDateTime** преобразует строку даты и времени с форматами, аналогичными предыдущим функциям.

При ошибках преобразования в рассмотренных функциях генерируется исключение **EConvertError**.

Функции из файла **dos.h** используют для хранения даты структуру типа **date**:

```
struct date{
    int da_year;      // текущий год
    char da_day;      // день месяца
    char da_mon;      // номер месяца (1 - январь)
};
```

Все данные хранятся в виде целых чисел, что облегчает их дальнейшую обработку. Приведем пример использования такой структуры. Следующие операторы создают структуру **D** типа **date** и заносят в нее текущую дату:

```
#include <dos.h>
...
struct date D;
getdate(&D);
```

В дальнейшем можно обращаться к полям этой структуры: **D.da_year**, **D.da_mon**, **D.da_day**.

Аналогичная структура предусмотрена и для хранения времени:

```
struct time {
    unsigned char ti_min;      // минуты
    unsigned char ti_hour;     // часы
    unsigned char ti_hund;     // сотые доли секунды
    unsigned char ti_sec;      // секунды
};
```

Функция **time** возвращает текущее время в секундах, отсчитанное от 0 часов 1 января 1970 по Гринвичу. Это время может быть преобразовано в строку с нулевым символом в конце, включающую год, месяц, день и т.д., с помощью функции **ctime** с учетом поправок на локальное время. Вид строки:

Mon Nov 21 11:31:54 1983\n\0

к сожалению, с английскими сокращениями.

Время, возвращаемое функцией **time**, может также с помощью функций **gmtime** (время по Гринвичу) или **localtime** (время с локальной поправкой) преобразовываться в поля структуры типа **tm**:

```
struct tm {
    int tm_sec;           // секунды
    int tm_min;           // минуты
    int tm_hour;          // часы (0 - 23)
    int tm_mday;          // день месяца (1 - 31)
    int tm_mon;           // месяц (0 - 11)
    int tm_year;          // год (календарный минус 1900)
    int tm_wday;          // день недели (0 - 6; 0 - воскресенье)
    int tm_yday;          // день года (0 - 365)
    int tm_isdst;         // установлен ли 12-часовой формат
};
```

Данная структура может быть преобразована в строку функцией **asctime**. Например, операторы:

```
time_t t = time(NULL);
struct tm *tt = localtime(&t);
char s[80];
strcpy(s, asctime(tt));
```

создают структуру типа **tm**, заносят в нее текущее время, полученное функцией **time** и преобразованное функцией **gmtime**, после чего формируют строку **s**. Но учтите, что строка получится аналогичной строке, возвращаемой описанной выше функцией **ctime** — т.е. использующей английские сокращения.

15.3.3 Функции преобразования типов

Функция	Синтаксис / Преобразует	Файл
Bounds	Windows::TRect Bounds(int ALeft, int ATop, int AWidth, int AHeight) Координаты ALeft и ATop и размеры AWidth и AHeight в TRect	Classes.hpp
Curr-ToFMTBCD	bool CurrToFMTBCD(System::Currency Curr, Bde::FMTBcd &BCD, int Precision, int Decimals) Значение Curr в тип Bde::FMTBcd	DBCommon.hpp
FMTBCDToCurr	bool FMTBCDToCurr(const Bde::FMTBcd &BCD, System::Currency &Curr) Значение BCD в тип Currency	DBCommon.hpp
Point	TPoint Point(int AX, int AY) Координаты AX и AY в TPoint	Classes.hpp
Rect	Windows::TRect Rect(int ALeft, int ATop, int ARight, int ABottom) Координаты ALeft , ATop , ARight , ABottom в TRect	Classes.hpp

Комментарии

Функции **CurrToFMTBCD** и **FMTBCDToCurr** осуществляют взаимное преобразование типа **Currency** и типа **Bde::FMTBcd**, используемого для хранения в полях BCD баз данных.

Функции **Bounds** и **Rect** осуществляют заполнение структуры типа **TRect**, хранящей информацию о местоположении и размерах прямоугольника и используемой во многих методах библиотеки компонентов. Определение этого типа см. в главе 16. Функция **Rect** задает значение переменной типа **TRect** координатами левой, верхней, правой и нижней границ - **ALeft**, **ATop**, **ARight**, **ABottom**. Функция **Bounds** задает значение переменной типа **TRect** координатами левого верхнего угла **ALeft** и **ATop**, шириной прямоугольника **AWidth** и его высотой **AHeight**.

В C++Builder 5 эти функции применяются, в частности, для задания значений таким свойствам компонентов, как **BoundsRect**, **ClientRect** и др. Например, приведенный ниже оператор размещает окно текстового редактора **Mem1** на его родительской панели **Panel1**, оставляя слева, внизу и справа зазор в 10 пикселей (для более приятного вида), а сверху — зазор 40 пикселей (например, для размещения заголовка окна):

```
Mem1->BoundsRect = Rect(10,40,Panel1->ClientWidth-10,
                        Panel1->ClientHeight-10);
```

То же самое можно сделать оператором:

```
Mem1->BoundsRect = Bounds(10,40,Panel1->ClientWidth-20,
                          Panel1->ClientHeight-50);
```

Функция **Point** осуществляет заполнение структуры типа **TPoint**, хранящей информацию о координатах точки и используемой во многих методах библиотеки компонентов. Определение этого типа см. в главе 16. Тип **TPoint** может также использоваться в конструкторе типа **TRect**, позволяющем описывать область двумя точками — левый верхний и правый нижний углы.

Ниже приведен ряд операторов, иллюстрирующих функции **Rect** и **Point**, а также применение типов **TRect** и **TPoint**:

```
TRect R, R1, R3;
R = Rect(10,100,20,200);
R1 = R;
```

```
TPoint P1, P2;
P1 = Point(10, 100);
P2 = Point(10, 200);
R3 = TRect(P1, P2);
```

```
TRect R2(P1, P2);
```

```
R.Left = 15;
int W = R.Width();
if(R1 != R) ...
```

15.4 Строки и символы

15.4.1 Функции обработки символов

Функция	Синтаксис / Описание	Файл
_tolower	int _tolower(int ch) Макрос приведения латинской буквы к нижнему регистру (без проверки)	ctype.h

Функция	Синтаксис / Описание	Файл
_toupper	int _toupper(int ch) Макрос приведения латинской буквы к верхнему регистру (без проверки)	ctype.h
isalnum	int isalnum(int c) Макрос проверки на латинскую букву или цифру	ctype.h
isalpha	int isalpha(int c) Макрос проверки на латинскую букву	ctype.h
isascii	int isascii(int c) Макрос проверки на символ из набора ASCII	ctype.h
isctrl	int isctrl(int c) Макрос проверки на управляющий символ	ctype.h
isdigit	int isdigit(int c) Макрос проверки на цифру	ctype.h
isgraph	int isgraph(int c) Макрос проверки на печатный символ (исключая пробел)	ctype.h
islower	int islower(int c) Макрос проверки на латинскую букву в нижнем регистре	ctype.h
isprint	int isprint(int c) Макрос проверки на печатный символ (включая пробел)	ctype.h
ispunct	int ispunct(int c) Макрос проверки на символ пунктуации (любой печатаемый, кроме латинской буквы, цифры, пробела)	ctype.h
isspace	int isspace(int c) Макрос проверки на пробельный символ (пробел, табуляция, новая строка)	ctype.h
isupper	int isupper(int c) Макрос проверки на латинскую букву в верхнем регистре	ctype.h
iswalnum	int iswalnum(wint_t c) Макрос проверки на латинскую букву или цифру	ctype.h
iswalpha	int iswalpha(wint_t c) Макрос проверки на латинскую букву	ctype.h
iswascii	int iswascii(wint_t c) Макрос проверки на символ из набора ASCII	ctype.h
iswctrl	int iswctrl(wint_t c) Макрос проверки на управляющий символ	ctype.h
iswdigit	int iswdigit(wint_t c) Макрос проверки на цифру	ctype.h
iswgraph	int iswgraph(wint_t c) Макрос проверки на печатный символ (исключая пробел)	ctype.h

Функция	Синтаксис / Описание	Файл
iswlower	int iswlower(wint_t c) Макрос проверки на латинскую букву в нижнем регистре	ctype.h
iswprint	int iswprint(wint_t c) Макрос проверки на печатный символ (включая пробел)	ctype.h
iswpunct	int iswpunct(wint_t c) Макрос проверки на символ пунктуации (любой печатаемый, кроме латинской буквы, цифры, пробела)	ctype.h
iswspace	int iswspace(wint_t c) Макрос проверки на пробельный символ (пробел, табуляция, новая строка)	ctype.h
iswupper	int iswupper(wint_t c) Макрос проверки на латинскую букву в верхнем регистре	ctype.h
iswxdigit	int iswxdigit(wint_t c) Макрос проверки на шестнадцатеричную цифру	ctype.h
isxdigit	int isxdigit(int c) Макрос проверки на шестнадцатеричную цифру	ctype.h
toascii	int toascii(int c) Макрос преобразования целого в код ASCII (очистка всех битов, кроме 7 младших) — в число от 0 до 127)	ctype.h
tolower	int tolower(int ch) Макрос приведения латинской буквы к нижнему регистру, если она в верхнем регистре	ctype.h
toupper	int toupper(int ch) Макрос приведения латинской буквы к верхнему регистру, если она в нижнем регистре	ctype.h
towlower	int tolower(wint_t ch) Макрос приведения латинской буквы к нижнему регистру, если она в верхнем регистре	ctype.h
towupper	int towupper(wint_t ch) Макрос приведения латинской буквы к верхнему регистру, если она в нижнем регистре	ctype.h

Комментарии

Обратите внимание на то, что макросы, распознающие и преобразовывающие буквы, не работают с символами кириллицы. Для кириллицы надо использовать работающие с кириллицей функции строк (см. разделы 15.4.2.2 и 15.4.2.3): перевести символ в строку, преобразовать строку и взять ее первый символ. Например, оператор

```
Key = AnsiUpperCase(Key) [1];
```

приведет символ **Key** типа **char** к верхнему регистру.

15.4.2 Функции обработки строк

15.4.2.1 Функции работы с областями памяти и строками

Функция	Синтаксис / Описание	файл
memcpy	void *memcpy(void *dest, const void *src, int c, size_t n) Копирует символы из src в dest , пока не встретится символ с или не будет скопировано n символов; возвращает dest	mem.h
memchr	void *memchr(const void *s, int c, size_t n) Возвращает указатель на первое вхождение символа c в первых n байтах s ; если символ не найден, возвращает NULL	mem.h
memcmp	int memcmp(const void *s1, const void *s2, size_t n) Сравнивает n символов из s1 и s2 ; результат < 0 при $s1 < s2$, $= 0$ при $s1 = s2$, > 0 при $s1 > s2$	mem.h
memcpy	void *memcpy(void *dest, const void *src, size_t n) Копирует n байтов из src в dest ; src и dest не должны перекрываться в памяти (см. memmove); возвращает dest	mem.h
memicmp	int memicmp(const void *s1, const void *s2, size_t n) Сравнивает, игнорируя регистр (не кириллицу), n символов из s1 и s2 ; результат < 0 при $s1 < s2$, $= 0$ при $s1 = s2$, > 0 при $s1 > s2$	mem.h
memmove	void *memmove(void *dest, const void *src, size_t n) Копирует n байтов из src в dest ; src и dest могут перекрываться в памяти (см. memcpy); возвращает dest	mem.h
memset	void *memset(void *s, int c, size_t n) Заполняет n байтов блока s символом c ; возвращает s	mem.h
setmem	void setmem(void *dest, unsigned length, char value) Заполняет блок dest размером length байтом value	mem.h

Комментарии

Приведенные в данном разделе функции могут работать как со строками с нулевым символом в конце, так и со строками без нулевого символа, а также с блоками памяти, не являющимися строками.

15.4.2.2 Функции обработки строк с нулевым символом в конце

Функция	Синтаксис / Описание	Файл
AnsiStrComp	int AnsiStrComp(char * S1, char * S2) Сравнивает строки S1 и S2 с учетом регистра; результат < 0 при $S1 < S2$, $= 0$ при $S1 = S2$, > 0 при $S1 > S2$	SysUtils.hpp

Функция	Синтаксис / Описание	Файл
AnsiStrIComp	int AnsiStrIComp(char * S1, char * S2) Сравнивает строки S1 и S2 без учета регистра; результат < 0 при S1 < S2 , = 0 при S1 = S2 , > 0 при S1 > S2	SysUtils.hpp
AnsiStrLComp	int AnsiStrLComp(char * S1, char * S2, Cardinal MaxLen) Сравнивает до MaxLen символов строк S1 и S2 ; результат < 0 при S1 < S2 , = 0 при S1 = S2 , > 0 при S1 > S2	SysUtils.hpp
AnsiStrLIComp	int AnsiStrLIComp(char * S1, char * S2, Cardinal MaxLen) Сравнивает до MaxLen символов строк S1 и S2 без учета регистра; результат < 0 при S1 < S2 , = 0 при S1 = S2 , > 0 при S1 > S2	SysUtils.hpp
AnsiStrLower	char * AnsiStrLower(char * Str) Возвращает строку, все символы которой приведены к нижнему регистру	SysUtils.hpp
AnsiStrPos	char * AnsiStrPos(char * Str, char * SubStr) Возвращает первое вхождение подстроки SubStr в Str или NULL	SysUtils.hpp
AnsiStrRScan	char * AnsiStrRScan(char * Str, char Chr) Возвращает указатель на последнее вхождение символа Chr в Str или NULL	SysUtils.hpp
AnsiStrScan	char * AnsiStrScan(char * Str, char Chr) Возвращает указатель на первое вхождение символа Chr в Str или NULL	SysUtils.hpp
AnsiStrUpper	char * AnsiStrUpper(char * Str) Возвращает строку, все символы которой приведены к верхнему регистру	SysUtils.hpp
CompareStr	int CompareStr(const System::AnsiString S1, const System::AnsiString S2) Сравнивает строки S1 и S2 с учетом регистра; результат < 0 при S1 < S2 , = 0 при S1 = S2 , > 0 при S1 > S2	SysUtils.hpp
CompareText	int CompareText(const System::AnsiString S1, const System::AnsiString S2) Сравнивает строки S1 и S2 без учета регистра; результат < 0 при S1 < S2 , = 0 при S1 = S2 , > 0 при S1 > S2	SysUtils.hpp
LineStart	char * LineStart(char * Buffer, char * BufPos) Возвращает указатель на начало последней строки в Buffer , кончающейся в позиции BufPos	SysUtils.hpp

Функция	Синтаксис / Описание	Файл
StrAlloc	char * StrAlloc(Cardinal Size) Динамически выделяет блок памяти под строку длиной Size - 1 и возвращает указатель на него; блок должен освобождаться функцией StrDispose	SysUtils.hpp
StrBufSize	Cardinal StrBufSize(char * Str) Возвращает максимальное число символов, которые могут разместиться в созданной функцией StrAlloc строке Str	SysUtils.hpp
strcat	char *strcat(char *dest, const char *src) Добавляет строку src в конец строки dest ; возвращает указатель на результирующую строку	string.h
StrCat	char * StrCat(char * Dest, char * Source) Добавляет строку Source в конец строки Dest ; возвращает указатель на результирующую строку	SysUtils.hpp
strchr	char *strchr(const char * s, int c) Возвращает указатель на первое вхождение c в s , или NULL	string.h
strcmp	int strcmp(const char *s1, const char *s2) Сравнивает строки s1 и s2 ; результат < 0 при $s1 < s2$, $= 0$ при $s1 = s2$, > 0 при $s1 > s2$	string.h
strcmpi	int strcmpi(const char *s1, const char *s2) То же, что strcmp : сравнивает строки s1 и s2 без учета регистра (не кириллицу); результат < 0 при $s1 < s2$, $= 0$ при $s1 = s2$, > 0 при $s1 > s2$	string.h
StrComp	int StrComp(char * Str1, char * Str2) Сравнивает строки S1 и S2 с учетом регистра (для кириллицы лучше использовать AnsiStrComp); результат < 0 при $S1 < S2$, $= 0$ при $S1 = S2$, > 0 при $S1 > S2$	SysUtils.hpp
StrCopy	char * StrCopy(char * Dest, char * Source) Копирует Source в Dest и возвращает Dest	SysUtils.hpp
strcpy	char *strcpy(char *dest, const char *src) Копирует строку src в dest ; возвращает dest	string.h
strcspn	size_t strcspn(const char *s1, const char *s2) Возвращает длину начальной части строки s1 , не содержащей ни одного из символов строки s2	string.h
strdup	char *strdup(const char *s) Выделяет соответствующую область в памяти и копирует в нее строку s ; возвращает указатель на эту область	string.h
StrECopy	char * StrECopy(char * Dest, char * Source) Копирует Source в Dest и возвращает указатель на конечный нулевой символ Dest	SysUtils.hpp

Функция	Синтаксис / Описание	Файл
StrEnd	char * StrEnd(char * Str) Возвращает указатель на конечный нулевой символ Str	SysUtils.hpp
strerror	char *strerror(int errnum) Возвращает указатель на строку сообщения об ошибке с номером errnum	string.h
stricmp	int stricmp(const char *s1, const char *s2) То же, что strcmpi : сравнивает строки s1 и s2 без учета регистра (не кириллицу); результат < 0 при s1 < s2 , = 0 при s1 = s2 , > 0 при s1 > s2	string.h
StrIComp	int StrIComp(char * Str1, char * Str2) Сравнивает строки S1 и S2 без учета регистра (для кириллицы надо использовать AnsiStrIComp); результат < 0 при S1 < S2 , = 0 при S1 = S2 , > 0 при S1 > S2	SysUtils.hpp
StrLCat	char * StrLCat(char * Dest, char * Source, Cardinal MaxLen) Копирует до MaxLen символов строки Source в конец строки Dest и возвращает Dest	SysUtils.hpp
StrLComp	int StrLComp(char * Str1, char * Str2, Cardinal MaxLen) Сравнивает до MaxLen символов строк S1 и S2 с учетом регистра (для кириллицы лучше использовать AnsiStrLComp); результат < 0 при S1 < S2 , = 0 при S1 = S2 , > 0 при S1 > S2	SysUtils.hpp
StrLCopy	char * StrLCopy(char * Dest, char * Source, Cardinal MaxLen) Копирует до MaxLen символов Source в Dest и возвращает указатель на Dest	SysUtils.hpp
strlen	size_t strlen(const char *s) Возвращает число символов в s , не считая нулевого символа в конце	string.h
StrLen	Cardinal StrLen(char * Str) Возвращает число символов в Str , не считая нулевого символа в конце	SysUtils.hpp
StrLIComp	int StrLIComp(char * Str1, char * Str2, Cardinal MaxLen) Сравнивает до MaxLen символов строк S1 и S2 без учета регистра (для кириллицы надо использовать AnsiStrLIComp); результат < 0 при S1 < S2 , = 0 при S1 = S2 , > 0 при S1 > S2	SysUtils.hpp
StrLower	char * StrLower(char * Str) Возвращает строку, все символы которой приведены к нижнему регистру (для кириллицы надо использовать AnsiStrLower)	SysUtils.hpp

Функция	Синтаксис / Описание	Файл
strlwr	char *strlwr(char *s) Преобразует строку s в нижний регистр (только латинские буквы)	string.h
StrMove	char * StrMove(char * Dest, char * Source, Cardinal Count) Копирует Count символов из Source в Dest и возвращает Dest ; Source и Dest могут перекрываться в памяти	SysUtils.hpp
strncat	char *strncat(char *dest, const char *src, size_t maxlen) Копирует до maxlen символов строки src в конец строки dest и добавляет нулевой символ; возвращает dest	string.h
strncmp	int strncmp(const char *s1, const char *s2, size_t maxlen) Сравнивает до maxlen символов строк s1 и s2 ; результат < 0 при s1 < s2 , = 0 при s1 = s2 , > 0 при s1 > s2	string.h
strncmpi	int strncmpi(const char *s1, const char *s2, size_t n) То же, что strnicmp : сравнивает до maxlen символов строк s1 и s2 без учета регистра (не кириллицу); результат < 0 при s1 < s2 , = 0 при s1 = s2 , > 0 при s1 > s2	string.h
strncpy	char *strncpy(char *dest, const char *src, size_t maxlen) Копирует до maxlen символов из src в dest ; возвращает dest	stdio.h
StrNew	char * StrNew(char * Str) Динамически размещает в памяти копию Str и возвращает указатель на нее	SysUtils.hpp
strnicmp	int strnicmp(const char *s1, const char *s2, size_t maxlen) То же, что strncmpi : сравнивает до maxlen символов строк s1 и s2 без учета регистра (не кириллицу); результат < 0 при s1 < s2 , = 0 при s1 = s2 , > 0 при s1 > s2	string.h
strnset	char *strnset(char *s, int ch, size_t n) Копирует символ ch в первые n символов s	string.h
strpbrk	char *strpbrk(const char *s1, const char *s2) Возвращает первое вхождение в s1 любого из символов строки s2 или NULL	string.h
StrPCopy	char * StrPCopy(char * Dest, const System::AnsiString Source) Копирует Source в Dest и возвращает Dest	SysUtils.hpp

Функция	Синтаксис / Описание	Файл
StrPLCopy	char * StrPLCopy(char * Dest, const System::AnsiString Source, Cardinal MaxLen) Копирует до MaxLen символов из Source в Dest и возвращает Dest	SysUtils.hpp
StrPos	char * StrPos(char * Str1, char * Str2) Возвращает первое вхождение подстроки Str2 в Str1 или NULL	SysUtils.hpp
strrchr	char *strrchr(const char *s, int c) Возвращает последнее вхождение символа c в s или NULL	string.h
strrev	char *strrev(char *s) Инвертирует (переворачивает) строку s кроме нулевого символа	string.h
StrRScan	char * StrRScan(char * Str, char Chr) Возвращает последнее вхождение символа Chr в Str или NULL	SysUtils.hpp
StrScan	char * StrScan(char * Str, char Chr) Возвращает первое вхождение символа Chr в Str или NULL	SysUtils.hpp
strset	char *strset(char *s, int ch); Заполняет всю строку s до нулевого символа символом ch	string.h
strspn	size_t strspn(const char *s1, const char *s2) Возвращает число первых символов строки s1 , входящих в множество символов строки s2 (последовательность символов безразлична)	string.h
strstr	char *strstr(const char *s1, const char *s2) Возвращает первое вхождение подстроки s2 в строку s1 или NULL	string.h
strtok	char *strtok(char *s1, const char *s2) Ищет первое вхождение разделителей из строки s2 в строке s1 и усекает строку s1 ; возможны повторные вызовы	string.h
StrUpper	char * StrUpper(char * Str) Возвращает строку, все символы которой приведены к верхнему регистру (для кириллицы надо использовать AnsiStrUpper);	SysUtils.hpp
strupr	char *strupr(char *s) Преобразует строку s в верхний регистр (только латинские буквы)	string.h

Комментарии

Функции файла **string.h**, распознающие регистр символов (**strempi**, **stricmp**, **strlwr**, **strncmpi**, **strnicmp**, **strupr**), не позволяют оперировать с символами кириллицы, записанными в разных регистрах. Для подобной работы с русскими текстами надо использовать аналогичные функции файла **SysUtils.hpp**. Функции этого файла могут работать с текстами на русском языке и с многобайтными символами.

Операции, выполняемые большинством функций, вероятно, понятны из пояснений в таблице. Поэтому остановимся только на некоторых из них.

Функция **strcat** прибавляет к тексту строки, указанной ее первым параметром, текст строки, указанной вторым параметром. Она возвращает указатель на строку, заданную ее первым параметром и содержащую суммарный текст обеих строк. Это позволяет делать вложенные вызовы **strcat**, если надо склеить несколько текстов. Функция **strcpy** копирует строку, являющуюся ее вторым параметром, в строку, являющуюся первым параметром и возвращает указатель на результат копирования. Функция **strstr** позволяет искать в строке некоторую заданную последовательность символов. Многочисленные примеры применения функций **strcat**, **strcpy**, **strstr** и **strlen** вы можете посмотреть в главе 13 в разделе 13.4.1.

Теперь рассмотрим функцию **strtok**, которая работает следующим образом. При своем первом вызове для данной строки **s1** функция ищет первое появление в строке одного из символов, содержащихся в строке **s2**. Если такой символ найден, то он заменяется на нулевой символ, т.е. строка **s1** усекается на этом символе. Функция возвращает указатель на первый символ усеченной строки **s1**. Далее можно повторно вызывать функцию **strtok**, задавая ей в качестве первого параметра **NULL**. Функция продолжит обработку той же строки **s1** (строку **s2** при этом можно сменить), найдет вхождение следующего символа из **s2**, опять заменит его нулевым символом и вернет указатель на начало нового просмотренного фрагмента строки. Таким образом, получается чтение строки по фрагментам. Приведем пример. Операторы

```
char s[80], *p;
...
p = strtok(s, " ,.");
if (p) Memo1->Lines->Add(p);
while(p)
{
    p = strtok(NULL, " ,.");
    if (p) Memo1->Lines->Add(p);
}
```

осуществляют поиск в строке **s** символов — разделителей: пробела, запятой, точки. Обработанные фрагменты строки заносятся в строки окна **Memo1**. Например, если в **s** занесен текст «Это текст строки, которая анализируется.», то приведенный код выдаст в окно **Memo1** строки:

```
Это
текст
строки
которая
анализируется
```

Если же мы уберем из второго параметра функции **strtok** символ пробела, то результатом будет:

```
Это текст строки
которая анализируется
```

Функция **LineStart** производит поиск в буфере **Buffer** назад от позиции **BufPos** символа конца строки «\n». Если символ найден, то функция возвращает указатель на него, выделяя таким образом последнюю строку. Обратившись к функции повторно и задав в качестве **BufPos** значение, на 1 меньшее возвращенно-

Функция	Синтаксис / Описание	Файл
AnsiUpperCase	System::AnsiString AnsiUpperCase(const System::AnsiString S) Возвращает строку S, приведенную к верхнему регистру (работает с кириллицей)	SysUtils.hpp
IsDelimiter	bool IsDelimiter(const System::AnsiString Delimiters, const System::AnsiString S, int Index) Определяет, является ли символ с индексом Index в строке S одним из разделителей, указанных в строке Delimiters	SysUtils.hpp
IsPathDelimiter	bool IsPathDelimiter(const System::AnsiString S, int Index); Определяет, является ли символ с индексом Index в строке S обратным слешем '\', используемым для задания путей к файлам	SysUtils.hpp
LastDelimiter	int LastDelimiter(const System::AnsiString Delimiters, const System::AnsiString S) Возвращает индекс последнего вхождения в строку S одного из разделителей, указанных в строке Delimiters	SysUtils.hpp
LowerCase	System::AnsiString LowerCase(const System::AnsiString S) Возвращает строку S, приведенную к нижнему регистру (для кириллицы используйте AnsiLowerCase)	SysUtils.hpp
QuotedStr	System::AnsiString QuotedStr(const System::AnsiString S) Возвращает строку S со вставленными в ее начало и конец символами одинарных кавычек и с заменой внутри строки одинарных кавычек на двойные (для многобайтных символов используйте AnsiQuotedStr)	SysUtils.hpp
StringRepase	System::AnsiString StringReplace(const System::AnsiString S, const System::AnsiString OldPattern, const System::AnsiString NewPattern, TReplaceFlags Flags) Возвращает строку S с заменой подстроки OldPattern на NewPattern ; Flags управляет заменами подстрок	SysUtils.hpp
Trim	System::AnsiString Trim(const System::AnsiString S) Возвращает строку S с удаленными начальными и конечными пробельными и управляющими символами	SysUtils.hpp

Функция	Синтаксис / Описание	Файл
TrimLeft	System::AnsiString TrimLeft(const System::AnsiString S) Возвращает строку S с удаленными начальными пробельными и управляющими символами	SysUtils.hpp
TrimRight	System::AnsiString TrimRight(const System::AnsiString S) Возвращает строку S с удаленными конечными пробельными и управляющими символами	SysUtils.hpp
UpperCase	System::AnsiString UpperCase(const System::AnsiString S) Возвращает строку S , приведенную к верхнему регистру (для кириллицы используйте AnsiUpperCase)	SysUtils.hpp
WrapText	System::AnsiString WrapText(const System::AnsiString Line, const System::AnsiString BreakStr, const TSysCharSet &BreakChars, int MaxCol) Возвращает текст Line , разбитый на строки длиной до MaxCol вставкой символов BreakStr и заменой на них символов множества BreakChars	SysUtils.hpp

Комментарии

Помимо функций, содержащихся в данной таблице, посмотрите в главе 16 описание класса **AnsiString**. В нем вы найдете много удобных методов работы со строками типа **AnsiString**.

Все функции, оперирующие со строками типа **AnsiString**, учитывают локализацию и поэтому могут с равным успехом работать как для латинских букв, так и для кириллицы. В этом их большое преимущество перед многими функциями, работающими со строками типа **char ***.

Несколько замечаний о приведенных в таблице функциях. В функциях **IsDelimiter** и **IsPathDelimiter** индексы отсчитываются от 1 (вопреки утверждениям встроенной справки C++Builder). Соответственно 1 — это первый символ строки, 2 — второй и т.д.

Функция **IsDelimiter** удобна для просмотра всех символов строки и замены каких-то одних символов на другие. Например, код

```
AnsiString S, Delimiters;
Delimiters = "'";
S = ...;
for(int i = 1; i <= StrLen(S.c_str()); i++)
    if(IsDelimiter(Delimiters, S, i))
        S[i] = ' ';
```

заменит в строке **S** все символы одинарных кавычек на двойные кавычки.

В этой функции в строке **Delimiters** не обязательно должны быть именно разделители. В нее могут быть занесены любые символы. Например, если приведенный код изменить следующим образом:

```
AnsiString S, Delimiters;
Delimiters = "123456789";
S = ...;
for(int i = 1; i <= StrLen(S.c_str()); i++)
```

```
if(IsDelimiter(Delimiters,S,i))
    S[i] -= 1;
```

то все символы цифр в строке, кроме 0, будут уменьшены на 1.

Функция **StringReplace** возвращает строку **S** с заменой подстроки **OldPattern** на **NewPattern**. Если параметр **Flags** не включает флаг **rfReplaceAll**, то функция заменят только первое вхождение подстроки **OldPattern**. Если параметр **Flags** включает флаг **rfIgnoreCase**, то операции выполняются без учета регистра. Например, оператор

```
S1 = StringReplace(S, OldPattern, NewPattern,
    TReplaceFlags() << rfReplaceAll);
```

поместит в строку **S1** текст строки **S** с заменой в ней всех подстрок **OldPattern** на **NewPattern**.

Функция **WrapText** разбивает заданный текст **Line** на строки. В качестве символов конца строки используются символы, заданные параметром **BreakStr**. Параметр **MaxCol** задает максимальное количество символов в строке. Разбиение на строки производится вставкой **BreakStr** после одного из символов, имеющих в множестве **BreakChars**. Вставка производится после того из символов в текущей строке, который обеспечивает ее максимальную длину в пределах **MaxCol**. Если ни одного символа из **BreakChars** не встретилось, длина строки может превысить **MaxCol**. Например, операторы

```
TSysCharSet bchars;
bchars << ' ' << '.' << ',' << ';' << '+' << '-';
AnsiString S, S1;
S = ...;
S1 = WrapText(S, "\n\r", bchars, 10);
```

обеспечивают запись в **S1** текста **S**, разбитого на строки длиной до 10 символов, причем разбиение проводится после пробелов и знаков пунктуации. Если в **S** записать текст: «Этот тест показывает разбиение на строки, в частности — на символах + и —», то результат будет следующим:

```
"Этот тест "
"показывает "
"разбиение "
"на строки,"
" в "
"частности "
"- на "
"символах +"
" и -."
```

Можно заметить, что вторая строка содержит 11 символов, включая пробел, т.е. ее размер больше заданного.

Конечно, в этом примере указана очень маленькая длина строки и поэтому разбиение выглядит не красиво. При нормальной для печати длине строк разбиение получается лучше.

15.5 Потоки и файлы

15.5.1 Атрибуты и флаги файлов, стандартные файлы

Файлы могут иметь следующие атрибуты, определенные в **dos.h**:

FA_RDONLY	только для чтения
FA_HIDDEN	невидимый

FA_SYSTEM	системный
FA_LABEL	метка тома
FA_DIREC	каталог
FA_ARCH	архивный

Еще один альтернативный набор констант атрибутов приведен в разделе 15.5.6.

Атрибуты объединяются в одно слово операцией ИЛИ (|).

При открытии файла доступ к нему определяется следующими флагами (определены в файле **fcntl.h**):

O_RDONLY	файл отрыт только для чтения
O_WRONLY	файл отрыт только для записи
O_RDWR	файл отрыт для чтения и записи
O_CREAT	создание нового файла
O_TRUNC	если файл существует, он урезается до 0
O_BINARY	двоичный файл
O_TEXT	текстовый файл
O_NOINHERIT	файл не передается в дочерний процесс
O_NDELAY	не используется, введен для совместимости с UNIX
O_APPEND	файл отрыт для добавления в конец, при каждой операции вывода указатель файла автоматически устанавливается на конец
O_CREA	если файл существует, то этот флаг не действует, если файл создается, то его флаги доступа задаются специальным параметром mode , принимающим значения, указанные в приведенной ниже таблице
O_EXCL	используется только вместе с O_CREA и означает, что, если файл уже существует, возвращается ошибка

Файлы могут открываться в следующих режимах **mode** (определены в файле **sys/stat.h**):

S_IWRITE	разрешение записи
S_IREAD	разрешение чтения
S_IREAD S_IWRITE	разрешение записи и чтения

Файлы могут иметь следующие флаги совместного доступа нескольких приложений (определены в файле **share.h**):

SH_COMPAT	Установка режима совместного доступа. Объединяется с другими флагами (например, SH_COMPAT SH_DENWR). Происходит ошибка, если файл уже открыт с другим режимом доступа
------------------	---

SH_DENWR	Запрещает запись, разрешает повторное открытие файла только для чтения
SH_DENYNO	Разрешает доступ для чтения и записи (оставлен наряду с SH_DENYNONE для обратной совместимости)
SH_DENYNONE	Разрешает доступ для чтения и записи. Разрешает повторное открытие файла, но только с тем же SH_COMPAT
SH_DENYRD	Запрещает чтение, разрешает повторное открытие файла только для записи
SH_DENYRW	Доступ к файлу обеспечивает только текущий дескриптор

Флаги могут соединяться в одно слово операцией ИЛИ (**|**). Из флагов **SH_DENYRD**, **SH_DENYNO** может быть задан только 1.

Имеется и другой набор констант режимов, в которых могут быть открыты файлы и которые определяют доступ к файлам других приложений (файл **SysUtils.hpp**):

Имя константы	Значение	Режим
fmOpenRead	\$0000	открыть только для чтения
fmOpenWrite	\$0001	открыть только для записи
fmOpenReadWrite	\$0002	открыть для чтения и записи
fmShareCompat	\$0000	открыть совместимым с FCB
fmShareExclusive	\$0010	запрет другим приложениям читать и записывать в файл
fmShareDenyWrite	\$0020	запрет другим приложениям записывать в файл
fmShareDenyRead	\$0030	запрет другим приложениям читать из файла
fmShareDenyNone	\$0040	полный доступ к файлу других приложений

В языках C и C++ файл рассматривается как поток (stream), представляющий собой последовательность считываемых или записываемых байтов.

В C++Builder могут использоваться два подхода к работе с файлами. Первый заключается в том, что информация о потоке (файле) заносится в структуру типа **FILE**, определенную в файле **stdio.h**, и файл оказывается связанным с этой структурой. Второй подход связывает файл с дескриптором (handle) — целым значением, характеризующим размещение информации о файле во внутренних таблицах системы.

В начале работы любой программы автоматически открывается три потока со своими дескрипторами:

Поток	Дескриптор	
stdin	0	стандартный входной поток — обычно клавиатура
stdout	1	стандартный выходной поток — обычно экран
stderr	2	стандартный поток сообщений об ошибках

В чистом виде эти потоки используются только в консольных приложениях. Но с помощью некоторых функций, описанных в последующих разделах, они могут быть перенаправлены в файлы и использоваться в этом случае в приложениях Windows.

15.5.2 Управление потоками и файлами, описываемыми структурами FILE

Управление файлами при подходе, описываемом структурами **FILE**, осуществляется следующими функциями.

Функция	Синтаксис / Описание	Файл
_fdopen	FILE *_fdopen(int handle, char * mode) Связывает файл с дескриптором handle , открываемый в режиме mode , с потоком и возвращает указатель на связываемую с потоком структуру типа FILE или NULL	stdio.h
_fileno	int _fileno(FILE *stream) Возвращает дескриптор потока stream	stdio.h
_flushall	int _flushall(void) Очищает буферы всех входных и выходных потоков, записывая в выходные потоки содержимое их буферов; возвращает число открытых и закрытых потоков	stdio.h
_fsopen	FILE *_fsopen(const char *filename, const char *mode, int shflag) Открывает файл filename совместного доступа, определяемого параметрами shflag и mode ; возвращает указатель на связываемую с ним структуру типа FILE или NULL	stdio.h, share.h
fclose	int fclose(FILE *stream) Закрывает поток stream	stdio.h
fflush	int fflush(FILE *stream) Очищает буфер выходного потока stream , сбрасывая его содержимое в поток; возвращает 0 при успешном завершении и EOF при ошибке	stdio.h
fopen	FILE *fopen(const char *filename, const char *mode) Открывает файл с именем filename в режиме mode и возвращает указатель на связываемую с ним структуру типа FILE или NULL	stdio.h
freopen	FILE *freopen(const char *filename, const char *mode, FILE *stream) Связывает с открытым потоком stream файл с именем filename в режиме mode и возвращает указатель на связываемую с ним структуру типа FILE или NULL	stdio.h
setbuf	void setbuf(FILE *stream, char *buf) Задаёт буфер buf для потока stream вместо буфера по умолчанию	stdio.h

Функция	Синтаксис / Описание	Файл
setvbuf	int setvbuf(FILE *stream, char *buf, int type, size_t size) Задаёт буфер buf размера size для потока stream вместо буфера по умолчанию	stdio.h
tmpfile	FILE *tmpfile(void) Открывает временный двоичный файл для записи и возвращает указатель на связываемую с ним структуру типа FILE или NULL	stdio.h

Комментарии

Открывается файл функцией **fopen**, в которую передается как параметр строка с именем файла **filename**. Аргумент **mode** указывает на строку, которая определяет режим открытия. Она может содержать спецификаторы:

r	открыть файл только для чтения
r+	открыть существующий файл для обновления — чтения и записи
a	открыть или создать файл для записи данных в конец файла
a+	открыть или создать файл для чтения или записи в конец файла
w	создать файл для записи; если такой файл уже существует, он будет перезаписан
w+	создать файл для обновления — чтения и записи; если такой файл уже существует, он будет перезаписан

К указанным спецификаторам в конце или перед символом «+» может добавляться символ «t» — текстовый файл, или «b» — бинарный, двоичный файл. Например, **rt**, **rb**, **r+t**, **r+b** и т.д. Если ни символ «t», ни символ «b» не указаны, то тип открываемого файла определяется значением глобальной переменной **_fmode**, определенной в файле **fcntl.h**. Она может принимать значения **O_TEXT** — текстовый файл (по умолчанию) или **O_BINARY** — двоичный файл.

Если открытие файла прошло успешно, функция **fopen** возвращает указатель на связываемую с потоком структуру типа **FILE**. Если произошла ошибка, то возвращается **NULL**. Типичная процедура открытия файла имеет вид:

```
FILE *F;
if ((F = fopen("Test.txt", "rt")) == NULL)
{
    ShowMessage("Файл не удастся открыть");
    return;
}
```

После того, как файл открыт, с ним связывается указатель, определяющий текущую позицию чтения и записи. При каждой операции чтения и записи этот указатель автоматически смещается на величину прочитанного или записанного поля. Кроме того указатель может смещаться программно с помощью функций **fseek** и **rewind**, описанных в разделе 15.5.4. Однако, если файл открыт для обновления — чтения и записи, то при работе с ним надо учитывать следующее:

- вывод (запись) не может следовать сразу за вводом (чтением) без предварительной установки указателя явным образом с помощью функций **fseek** или **rewind**
- ввод не может следовать сразу за выводом без предварительной установки указателя явным образом с помощью функций **fseek** или **rewind** — в противном случае ввод может неожиданно выдать признак конца файла

Функция **_fsopen** открывает файл **filename** совместного доступа для нескольких процессов, определяемого параметрами **shflag** и **mode**. Флаги совместного доступа, из которых может формироваться **shflag**, см. в разделе 15.5.1. Параметр **mode** аналогичен такому же параметру функции **fopen**. При работе с этой функцией в DOS предварительно должна быть загружена программа **SHARE.EXE**.

Функция **freopen** связывает с открытым потоком **stream** файл с именем **filename** в режиме **mode**. Отличие от функции **fopen** заключается только в том, что поток **stream** уже был ранее открыт. Это позволяет, в частности, переназначить стандартный поток. Например, оператор

```
FILE *F = freopen("output.txt", "wt", stdout);
```

перенаправляет стандартный выходной поток **stdout** в текстовый файл «output.txt». Все последующие выводы в поток **stdout** будут в действительности направляться в этот файл.

Функция **_fdopen** осуществляет связь между файлами, открытыми с дескрипторами (об этом подходе см. в разделе 15.5.3), и потоками типа **FILE**. Например, оператор

```
FILE * stream = fdopen(handle, "w");
```

связывает ранее открытый файл с дескриптором **handle** со структурой потока **stream**.

Обратное преобразование — получение дескриптора файла, связанного со структурой **FILE**, осуществляет функция **_fileno**. В приведенном ниже примере создается файл «Test.txt» для записи и определяется его дескриптор.

```
FILE *stream;
int handle;
stream = fopen("Test.txt", "w"); // создание файла
handle = _fileno(stream);        // определение дескриптора
...
fclose(stream);                  // закрытие файла
```

Открываемые рассмотренными функциями потоки буферизуются, т.е. обмен информацией происходит не непосредственно с файлами, а с промежуточными буферами, расположенными в оперативной памяти. Информация переписывается из буфера в файл только при переполнении буфера, или при закрытии файла, или функциями **fflush** и **_flushall**. Первая из них действует на буфер указанного выходного потока, вторая — на буферы всех входных и выходных потоков. Для входного потока функция очищает буфер, а для выходного — немедленно сбрасывает все содержимое в поток, после чего буфер очищается. Потоки остаются открытыми и буферы готовы к приему новой информации.

Программа автоматически осуществляет буферизацию всех потоков. Однако, этим процессом можно управлять функциями **setbuf** и **setvbuf**. Если в функции **setbuf** параметр **buf** задать равным **NULL**, то буферизация потока производиться не будет. Это может замедлить работу с потоком, но зато обеспечит немедленную передачу информации без ожидания того, чтобы буфер переполнился. Если же **buf** указывает на массив символов, то именно этот массив будет использоваться для полной буферизации потока **stream** вместо буфера по умолчанию. Размер буфера может достигать значения **BUFSIZ**, определенного в файле **stdio.h**. Например:

```
char outbuf[BUFSIZ];
setbuf(F, outbuf);
```

Функция **setbuf** должна вызываться сразу после открытия потока или сразу после вызова функции **fseek** (см. раздел 15.5.4), устанавливающей позицию указателя потока. В противном случае вызов **setbuf** может приводить к непредсказуемым результатам.

Функция **setvbuf** предоставляет более богатые возможности по управлению процессом буферизации. В этой функции задание параметра **buf = NULL** приводит к выделению в динамически распределяемой памяти с помощью функции **malloc** места для буфера размером **size**. Этот буфер автоматически освобождает память при закрытии соответствующего потока. Размер открываемого буфера ограничен сверху константой **UINT_MAX**, определенной в файле **limits.h**.

Параметр **type** может принимать следующие значения:

_IOFBF	Полная буферизация файла. Когда буфер ввода пуст, очередная операция ввода пытается заполнить весь буфер. При выводе выдача содержимого в файл производится после того, как буфер заполнится до отказа
_IOLBF	Буферизация строк. Когда буфер ввода пуст, очередная операция ввода пытается, как и в предыдущем случае, заполнить весь буфер. Однако при выводе выдача содержимого в файл производится после того, как в потоке появляется символ новой строки
_IONBF	Небуферизованный ввод/вывод. При этом параметры buf и size игнорируются

При успешном завершении функция **setvbuf** возвращает 0.

Файлы, открытые рассмотренными ранее функциями **fopen** и **freopen**, должны закрываться функцией **fclose**. При этом автоматически открытые буферы потоков освобождают память. Но если буферы назначались явным образом функциями **setbuf** и **setvbuf** с параметром **buf** отличным от **NULL**, то они автоматически не освобождают память.

Функция **tmpfile** открывает временный двоичный файл для записи в режиме **(w+b)** и возвращает указатель на связываемую с ним структуру типа **FILE**. При неудаче возвращается **NULL**. Если после создания временного файла программа не изменяет текущий каталог, то при завершении программы временный файл автоматически удаляется с диска.

Подробное рассмотрение работы с файлами, описываемыми структурами **FILE**, см. в главе 13 в разделе 13.9.2.

15.5.3 Управление потоками и файлами, связанными с дескрипторами

Управление файлами осуществляется следующими функциями.

Функция	Синтаксис / Описание	Файл
_creat	int _creat(const char *path, int mode) Создает новый или переписывает существующий файл в режиме mode ; возвращает дескриптор или -1	io.h , sys\stat.h
_rtl_close	int _rtl_close(int handle) Закрывает поток с дескриптором handle	io.h
_rtl_creat	int _rtl_creat(const char *path, int attrib) Создает новый или переписывает существующий файл path с атрибутами attrib ; возвращает дескриптор или -1	io.h , dos.h

Функция	Синтаксис / Описание	Файл
_rtl_open	int _rtl_open(const char *filename, int oflags) Открывает существующий файл filename для чтения или записи с атрибутами oflags ; возвращает дескриптор или -1	io.h, dos.h
_sopen	int _sopen(char *path, int access, int shflag [, int mode]) Открывает файл path совместного доступа, определяемого параметрами access , shflag , mode ; возвращает дескриптор или -1	fcntl.h, sys/stat.h, share.h, io.h, stdio.h
close	int close(int handle) Закрывает поток с дескриптором handle	io.h
creatnew	int creatnew(const char *path, int attrib) Аналогична _rtl_creat , но выдает ошибку, если файл существует	io.h, dos.h
creattemp	int creattemp(char *path, int attrib) Создает временный файл с уникальным именем и атрибутами attrib в каталоге path	io.h, dos.h
dup	int dup(int handle) Создает и возвращает дубликат дескриптора handle	io.h
dup2	int dup2(int oldhandle, int newhandle) Создает и возвращает дубликат newhandle дескриптора oldhandle	io.h
FileClose	void FileClose(int Handle) Закрывает файл с дескриптором Handle	SysUtils.hpp
FileCreate	int FileCreate(const System::AnsiString FileName) Создает файл FileName и возвращает его дескриптор в случае успеха или -1	SysUtils.hpp
FileOpen	int FileOpen(const System::AnsiString FileName, int Mode) Открывает файл FileName в режиме Mode и возвращает его дескриптор или -1	SysUtils.hpp
lock	int lock(int handle, long offset, long length) Блокирует в файле handle length байтов, начиная с позиции offset от чтения или записи другими процессами	io.h
locking	int locking(int handle, int cmd, long length) Блокирует или разблокирует в файле handle length байтов, начиная с текущей позиции для доступа других процессов	io.h, sys/locking.h
setmode	int setmode(int handle, int amode) С помощью параметра amode задает и возвращает тип открытого файла с дескриптором handle : O_BINARY — двоичный, O_TEXT — текстовый	io.h

Функция	Синтаксис / Описание	Файл
umask	unsigned umask(unsigned mode) Задает маску режима чтения/записи mode , принимаемую по умолчанию функциями open и creat : S_IWRITE , S_IREAD или S_IREAD S_IWRITE ; возвращает предыдущую маску	io.h
unlock	int unlock(int handle, long offset, long length) Разблокирует в файле handle length байтов, начиная с позиции offset , для чтения или записи другими процессами	io.h

Комментарии

Функция **_creat** создает новый или переписывает существующий файл в режиме **mode**. Параметр **path** указывает имя файла или имя с путем к нему. Вид файла — текстовый или двоичный, задается глобальной переменной **_fmode** — **O_TEXT** или **O_BINARY**. Если файл существует и для него установлен атрибут записи, то длина файла усекается до 0. Если же файл существует и имеет атрибут только для чтения, то функция **_creat** выдает ошибку, а файл сохраняется неизменным.

Режим, в котором создается файл, определяется параметром **mode**, который может принимать значения, определенные в файле **sys\stat.h** и указанные в разделе 15.5.1.

При успешном завершении возвращается дескриптор созданного файла. При ошибке возвращается -1, а значение **errno** (см. раздел 15.1.5.1) может иметь значения **EACCES**, **ENOENT**, **EMFILE**.

В настоящее время функция **_creat** считается устаревшей и вместо нее рекомендуется использовать **_rtl_creat**. Она действует подобно функции **_creat**, но всегда создает двоичный файл и позволяет своим параметром **attrib** установить операцией ИЛИ (|) атрибуты: **FA_RDONLY** — только для чтения, **FA_HIDDEN** — невидимый, **FA_SYSTEM** — системный (подробнее об атрибутах см. в разделе 15.5.1).

Функция **creatnew** аналогична функции **_rtl_creat** во всем, кроме того, что возвращает -1 в случае, если файл с данным именем уже существует.

Функция **_rtl_open** открывает существующий файл **filename** для чтения или записи с атрибутами **oflags**. Таблица возможных атрибутов приведена в разделе 15.5.1. При успешном завершении возвращается дескриптор открытого файла и его указатель устанавливается на начало файла. При ошибке возвращается -1, а значение **errno** (см. раздел 15.1.5.1) может иметь значения **EINVA**CC, **EACCES**, **ENOENT**, **EMFILE**.

Функция **_sopen** открывает файл **path** совместного доступа, определяемого параметрами **access**, **shflag**, **mode**. Параметр **access** задает совокупность **O_...** флагов доступа, перечисленных в разделе 15.5.1. Если среди этих флагов задан **O_CREA**, то режим открытия файла определяется параметром **mode** (см. раздел 15.5.1). Параметр **shflag** определяет флаги совместного доступа к файлу нескольких приложений. Значения этих флагов приведены в разделе 15.5.1. При успешном завершении функция возвращает дескриптор открытого файла и его указатель устанавливается на начало файла. При ошибке возвращается -1, а **errno** (см. раздел 15.1.5.1) может принимать значения **EINVA**CC, **EACCES**, **ENOENT**, **EMFILE**.

Функция **creattemp** создает временный файл с уникальным именем и атрибутами **attrib** в каталоге **path**. Вид файла — текстовый или двоичный, определяется значением глобальной переменной **_fmode** (**O_TEXT** или **O_BINARY**). Параметр **attrib** может равняться нулю или принимать уже рассмотренные значения **FA_HIDDEN**, **FA_RDONLY** или **FA_SYSTEM** (см. раздел 15.5.1).

Число файлов одновременно открытых перечисленными функциями, не должно превышать **HANDLE_MAX**.

Функции **close**, **_rtl_close** и **FileClose** закрывают файл, открытый ранее функциями **creat**, **creatnew**, **creattemp**, **dup**, **dup2**, **open**, **_rtl_creat**, **_rtl_open**, **FileOpen**. При этом в выходной файл не записывается автоматически признак конца файла Ctrl-Z. Если этот символ требуется, вам надо предварительно записать его явным образом.

При успешном завершении функций они возвращают 0. При ошибке возвращают значение -1 и задают глобальной переменной **errno** (см. раздел 15.1.5.1) значение **EBADF**.

Таким образом, стандартная схема работы с файлами, связанными с дескрипторами, следующая:

```
int hout = open("output.txt", O_CREAT | O_WRONLY, S_IWRITE);
...
close(hout);
```

Например, операторы

```
int handle;
if ((handle = open("Test.txt", O_CREAT | O_TEXT)) == -1)
{
    ShowMessage("Файл не удастся создать");
    return;
}
...
close(handle);
```

пытаются создать новый текстовый файл, а в случае неудачи (функция **open** вернула -1) отображают сообщение об ошибке.

Функции **FileOpen**, **FileCreate** и **FileClose** дают альтернативный подход к открытию и закрытию файлов, связанных с дескрипторами. Функция **FileOpen** открывает файл в режиме **Mode**, задаваемом константами **fmShare** (см. раздел 15.5.1). В дальнейшем с этими файлами можно работать с помощью функций **FileRead**, **FileWrite**, **FileSeek**, описанных в разделе 15.5.4.

При совместном доступе к файлам нескольких приложений помимо установки флагов доступа может использоваться блокировка и деблокировка отдельных областей файла с помощью функций **lock**, **unlock**, **locking**. При работе с этими функциями в DOS предварительно должна быть загружена программа **SHARE.EXE**. В функции **locking** режим работы определяется параметром **cmd**:

LK_LOCK	Блокировать область. Если не удалось, то прежде, чем отказаться от блокировки, делается новая попытка через 10 секунд
LK_RLCK	То же, что LK_LOCK
LK_NBLCK	Блокировать область. Если не удалось, то происходит отказ от попытки блокировки
LK_NBRLCK	То же, что LK_NBLCK
LK_UNLCK	Разблокировать ранее заблокированную область файла

При успешном завершении функции возвращается 0. При неудаче возвращается -1, а значение **errno** (см. раздел 15.1.5.1) может иметь значения **EACCES**, **EBADF**, **EDEADLOCK** — невозможность блокировки, несмотря на повторную попытку через 10 секунд (при **cmd** равном **LK_LOCK** или **LK_RLCK**), **EINVAL** — ошибка в **cmd** или не загружена программа **SHARE.EXE**.

Функции **dup** и **dup2** позволяют оперировать с дескрипторами файлов. Функция **dup** создает и возвращает дубликат (псевдоним) дескриптора **handle**. Дубликат связан с тем же открытым файлом или устройством, что и исходный дескриптор, имеет тот же режим доступа (только чтение, только запись, чтение и запись) и имеет тот же указатель позиции в файле. Изменение указателя в одном из дескрипторов приводит к синхронному сдвигу указателя в другом дескрипторе. Функция **dup2** создает и возвращает аналогичный функции **dup** дубликат **newhandle** дескриптора **oldhandle**. Функции **dup** и **dup2** могут использоваться, в частности, для перенаправления стандартных потоков. Например, операторы

```
int hout = open("output.txt", O_CREAT | O_WRONLY, S_IWRITE);
dup2(hout, 1);
```

перенаправляют стандартный выходной поток **stdout** (его дескриптор равен 1 — см. раздел 15.5.1) в файл «output.txt».

15.5.4 Функции ввода/вывода

Функция	Синтаксис / Описание	Файл
_fgetchar	int _fgetchar(void) Вводит символ из потока stdin	stdio.h
_fputchar	int _fputchar(int c) Выводит символ c в поток stdout , то же, что fputc(c, stdout) ; при ошибке возвращает EOF	stdio.h
_getw	int _getw(FILE *stream) Вводит целое число из потока stream	stdio.h
cgets	char *cgets(char *str) Читает строку символов с консоли	conio.h
clearerr	void clearerr(FILE *stream) Очищает индикаторы ошибок и конца файла потока stream	stdio.h
cprintf	int cprintf(const char *format [, argument, ...]) Выводит на экран список аргументов argument по формату format (см. раздел 15.1.4.1)	conio.h
cputs	int cputs(const char *str) Выводит строку на экран; возвращает последний символ	conio.h
cscanf	int cscanf(char *format [, address, ...]) Вводит данные с консоли в список аргументов по адресам argument по формату format ; возвращает число успешно введенных полей или EOF при конце файла	conio.h
eof	int eof(int handle) Возвращает 0 при достижении конца потока (файла), связанного с дескриптором handle	io.h

Функция	Синтаксис / Описание	Файл
fEOF	int fEOF(FILE *stream) Возвращает 0 при достижении конца потока (файла) stream	stdio.h
ferror	int ferror(FILE *stream) Проверяет ошибки ввода/вывода потока stream и возвращает 0 при отсутствии ошибки	stdio.h
fgetc	int fgetc(FILE *stream) Вводит символ из потока stream и возвращает его обратно	stdio.h
fgetpos	int fgetpos(FILE *stream, fpos_t *pos) Заносит в pos текущую позицию файла stream ; при успехе возвращает 0	stdio.h
fgets	char *fgets(char *s, int n, FILE *stream) Вводит в s и возвращает строку до n символов из потока stream	stdio.h
FileRead	int FileRead(int Handle, void *Buffer, int Count) Выводит из файла с дескриптором Handle , открытого функциями FileOpen или FileCreate , Count байтов в буфер Buffer ; возвращает число прочитанных байтов или -1	SysUtils.hpp
FileSeek	int FileSeek(int Handle, int Offset, int Origin) Перемещает указатель файла с дескриптором Handle , открытого функциями FileOpen или FileCreate , на Offset байтов от позиции Origin ; возвращает 0 при успешном завершении	SysUtils.hpp
FileWrite	int FileWrite(int Handle, const void *Buffer, int Count) Вводит в файл с дескриптором Handle , открытый функциями FileOpen или FileCreate , Count байтов из буфера Buffer ; возвращает число записанных байтов или -1	SysUtils.hpp
fprintf	int fprintf(FILE *stream, const char *format [, argument, ...]) Выводит в файл stream список аргументов argument по формату format ; возвращает число успешно записанных байтов или EOF при ошибке	stdio.h
fputc	int fputc(int c, FILE *stream) Выводит символ c в поток stream	stdio.h
fputs	int fputs(const char *s, FILE *stream) Выводит строку s в поток stream ; при ошибке возвращает EOF	stdio.h

Функция	Синтаксис / Описание	Файл
fread	size_t fread(void *ptr, size_t size, size_t n, FILE *stream) Неформатированное чтение из stream в ptr n элементов данных размером size каждый; возвращает число успешно прочитанных байтов (n * size)	stdio.h
fscanf	int fscanf(FILE *stream, const char *format [, address, ...]) Вводит данные из файла stream в список аргументов по адресам argument по формату format ; возвращает число успешно введенных полей или EOF при конце файла	stdio.h
fseek	int fseek(FILE *stream, long offset, int fromwhere) Перемещает указатель файла stream на offset байтов от позиции fromwhere ; возвращает 0 при успешном завершении	stdio.h
fsetpos	int fsetpos(FILE *stream, const fpos_t *pos) Устанавливает указатель потока stream в позицию pos	stdio.h
ftell	long int ftell(FILE *stream) Возвращает текущую позицию файла stream	stdio.h
fwrite	size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream) Неформатированная запись из ptr в stream n элементов данных размером size каждый; возвращает число успешно записанных байтов (n * size)	stdio.h
getc	int getc(FILE *stream) Вводит символ из потока stream	stdio.h
getch	int getch(void) Вводит символ с консоли без эхо на экране	conio.h
getchar	int getchar(void) Вводит символ из stdin ; то же, что getc(stdin)	stdio.h
getche	int getche(void) Вводит символ с консоли с эхо на экране	conio.h
getpass	char *getpass(const char *prompt) Вводит пароль с консоли до восьми символов после печати на экране приглашения prompt ; возвращает введенную строку	conio.h
gets	char *gets(char *s) Вводит строку из stdin	stdio.h
kbhit	int kbhit(void) Проверяет нажатие клавиши консоли; если ни одна клавиша не нажата — возвращает 0	conio.h

Функция	Синтаксис / Описание	Файл
lseek	long lseek(int handle, long offset, int fromwhere) Перемещает указатель файла с дескриптором handle на offset байтов от позиции fromwhere ; возвращает 0 при успешном завершении	io.h
perror	void perror(const char *s) Выводит в стандартный выходной поток ошибок сообщение s	stdio.h
printf	int printf(const char *format [, argument, ...]) Выводит в стандартный поток stdout список аргументов argument по формату format (см. раздел 15.1.4.1)	stdio.h
putc	int putc(int c, FILE *stream) Выводит символ c в поток stream	stdio.h
putch	int putch(int c) Выводит символ c на экран	conio.h
putchar	int putchar(int c) Макрос, выводящий символ c в stdout ; эквивалентен putc(c, stdout)	stdio.h
puts	int puts(const char *s) Выводит строку s в stdout и добавляет символ новой строки	stdio.h
puttext	int puttext(int left, int top, int right, int bottom, void *source) Выводит содержимое блока памяти source на экран в текстовом режиме в прямоугольник с координатами left , top , right , bottom	conio.h
putw	int _putw(int w, FILE *stream) Выводит в поток stream целое w	stdio.h
read	int read(int handle, void *buf, unsigned len) Чтение из файла с дескриптором handle в буфер buf len байтов	io.h
scanf	int scanf(const char *format [, address, ...]) Вводит данные из потока stdin в список аргументов по адресам argument по формату format (см. раздел 15.1.4.1)	stdio.h
sprintf	int sprintf(char *buffer, const char *format [, argument, ...]) Выводит в строку buffer список аргументов argument по формату format (см. раздел 15.1.4.2)	conio.h
sscanf	int sscanf(const char *buffer, const char *format [, address, ...]) Вводит данные из строки buffer в список аргументов по адресам argument по формату format	conio.h

Функция	Синтаксис / Описание	Файл
tell	long tell(int handle) Возвращает текущую позицию файла с дескриптором handle	io.h
ungetc	int ungetc(int c, FILE *stream) Возвращает символ ch в поток stream , чтобы он стал следующим символом для чтения	stdio.h
ungetch	int ungetch(int ch) Возвращает символ ch на консоль, чтобы он стал следующим символом для чтения	conio.h
fprintf	int fprintf(FILE *stream, const char *format, va_list arglist) Выводит в файл stream список аргументов arglist по формату format (см. раздел 15.1.4.1)	stdio.h
vfscanf	int vfscanf(FILE *stream, const char *format, va_list arglist) Вводит данные из потока stream в список адресов аргументов arglist по формату format (см. раздел 15.1.4.2)	stdio.h
vprintf	int vprintf(const char *format, va_list arglist) Выводит в stdout список аргументов arglist по формату format (см. раздел 15.1.4.1)	stdarg.h
vscanf	int vscanf(const char *format, va_list arglist) Вводит данные из потока stdin в список адресов аргументов arglist по формату format (см. раздел 15.1.4.2)	stdarg.h
vsprintf	int vsprintf(char *buffer, const char *format, va_list arglist) Выводит в строку buffer список аргументов arglist по формату format (см. раздел 15.1.4.1)	stdarg.h
vsscanf	int vsscanf(const char *buffer, const char *format, va_list arglist) Вводит данные из буфера buffer в список адресов аргументов arglist по формату format (см. раздел 15.1.4.2)	stdarg.h
write	int write(int handle, void *buf, unsigned len) Выводит в файл с дескриптором handle из буфера buf len байтов	io.h

Комментарии

Для работы с текстовыми файлами чаще всего используются функции **scanf** для чтения и **printf** для записи.

Функции **printf**, **sprintf**, **fprintf**, **vprintf**, **vsprintf** производят форматированный вывод данных из списка указанных в них аргументов. Строка форматирования подробно рассмотрена в разделе 15.1.4.1. Функции возвращают число успешно записанных байтов или EOF при ошибке.

Функции **fscanf**, **scanf**, **cscanf**, **sscanf**, **vfscanf**, **vscanf**, **vsscanf** производят форматированный ввод данных в список адресов аргументов. Строка форматирования подробно рассмотрена в разделе 15.1.4.2. Число спецификаций в строке форматирования и число аргументов в списке должны совпадать. Функции возвращают число успешно введенных полей, а если достигнут конец файла, то возвращают **EOF**. Обратите внимание на то, что указываются не сами переменные, в которые осуществляется ввод данных, а их адреса. Например:

```
FILE *F;
if ((F = fopen("Test2.txt", "rt")) == NULL)
{
    ShowMessage("Файл не удается открыть");
    return;
}
int i1, i2;
double r;
if (fscanf(F, "%d%d%le", &i1, &i2, &r) != 3)
    ShowMessage("Ошибка чтения");
else
{
    ...
}
fclose(F);
```

Функции **vfprintf** и **vfscanf** работают со списком аргументов переменной длины типа **va_list**, который объявлен в файле **stdarg.h** (см. раздел 15.7.4). Они могут использоваться внутри функций, принимающих переменное число параметров. Например:

```
#include <stdio.h>
#include <stdarg.h>

FILE *fp;
...

void pr(char *format, ...)
{
    fp = tmpfile();
    if (fp == NULL)
    {
        ShowMessage("Временный файл не может быть создан");
        exit(1);
    }
    va_list ap;
    int arg;
    va_start(ap, format);
    arg = vfprintf(fp, format, ap);
    va_end(ap);
}
```

Вызов такой функции может иметь, например, вид:

```
pr("%d %d %d", 10, 20, 30);
```

Функции **getchar**, **_fgetchar**, **cgets**, **cprintf**, **_fputchar**, **putch**, **puttext**, **getch**, **getche**, **cputs**, **getpass**, **cscanf**, **kbhit**, **ungetc**, **ungetch**, **vprintf**, **vscanf** не могут использоваться в приложениях с графическим интерфейсом для Win32.

В приложениях с графическим интерфейсом для Win32 при использовании функций **scanf**, **gets** должен быть перенаправлен поток **stdin**, а при использовании функций **printf**, **putchar**, **puts** должен быть перенаправлен поток **stdout**.

Функции **getchar**, **_fgetchar**, **getc**, **fgetc** возвращают читаемый символ, преобразованный в целое без знака.

Функции **getchar**, **_fgetchar**, **getc**, **gets**, **fgets**, **fgetc**, **fputc** при ошибке преобразования или при окончании файла возвращают **EOF**.

Функции **getc**, **fgetc**, **getchar** после чтения возвращают символ в поток. При этом функции **getc** увеличивают указатель потока на 1, подготавливая чтение следующего символа.

Функция **_getw** возвращает целое, прочитанное из потока. Поток (файл) должен быть открыт в текстовом режиме. Функции **putw**, **puts** записывают соответственно целое и строку в поток. При ошибке преобразования или при окончании файла все эти функции возвращают **EOF**. Поскольку **EOF** является допустимым возвращаемым значением, для проверки конца файла или ошибки преобразования надо использовать функции **feof** и **ferror**.

Функция **gets** читает последовательность символов до символа конца строки, который не помещает в возвращаемую строку, заменяя его нулевым символом.

Функция **fgets** читает последовательность символов до заданного числа символов **n** или до символа конца строки, который помещает в возвращаемую строку, помещая после него нулевой символ.

Функция **puttext**, используемая только в консольных приложениях, выводит содержимое блока памяти, на который указывает **source**, на экран в текстовом режиме в прямоугольник с координатами **left**, **top**, **right**, **bottom**. Координаты левого верхнего угла (1,1). Каждой позиции на экране соответствуют 2 байта, первый из которых — символ, а второй — атрибуты вывода. Функция возвращает ненулевое значение при успешном выводе и 0 — при ошибке.

Функции **fseek** и **lseek** перемещают указатель файла на **offset** байтов от позиции **fromwhere**. Для текстового файла параметр **offset** должен быть равен 0 или соответствовать допустимому значению, возвращенному функцией **ftell**. Параметр **fromwhere**, определяющий точку, относительно которой производится смещение **offset**, может принимать значения:

SEEK_SET	0	начало файла
SEEK_CUR	1	текущая позиция файла
SEEK_END	2	конец файла

Функции возвращают 0 при успешном завершении.

Функции **FileRead**, **FileWrite** и **FileSeek** используются для работы с файлами, открытыми функциями **FileOpen** или **FileCreate**.

15.5.5 Функции обработки имен файлов

Функция	Синтаксис / Описание	Файл
_mktemp	char *_mktemp(char *template) Генерирует, заносит в template и возвращает уникальное имя файла, которое может в дальнейшем использоваться для создания временных файлов	dir.h
ChangeFileExt	System::AnsiString ChangeFileExt(const System::AnsiString FileName, const System::AnsiString Extension) Возвращает имя файла FileName с измененным расширением на Extension ; сам файл не переименовывается	SysUtils.hpp

Функция	Синтаксис / Описание	Файл
ExpandFileName	System::AnsiString ExpandFileName(const System::AnsiString FileName) Расширяет имя файла FileName , добавляя к нему текущие путь и диск; проверка существования такого файла не проводится	SysUtils.hpp
ExpandUNCFileName	System::AnsiString ExpandUNCFileName(const System::AnsiString FileName) Расширяет имя файла FileName , добавляя к нему текущие путь и том в формате UNC: «\\<servername>\<sharename>», если том указывает на сеть	SysUtils.hpp
ExtractFileDir	System::AnsiString ExtractFileDir(const System::AnsiString FileName) Извлекает из FileName и возвращает путь к файлу	SysUtils.hpp
ExtractFileDrive	System::AnsiString ExtractFileDrive(const System::AnsiString FileName) Возвращает диск файла FileName (например, «с:») или в формате UNC: «\\<servername>\<sharename>», если путь указывает на сеть	SysUtils.hpp
ExtractFileExt	System::AnsiString ExtractFileExt(const System::AnsiString FileName) Возвращает расширение файла FileName	SysUtils.hpp
ExtractFileName	System::AnsiString ExtractFileName(const System::AnsiString FileName) Возвращает имя файла, извлеченное из FileName , т.е. конец строки после последнего обратного следа или двоеточия	SysUtils.hpp
ExtractFilePath	System::AnsiString ExtractFilePath(const System::AnsiString FileName) Возвращает путь к файлу, извлеченный из FileName , включая последний обратный слеш или двоеточие, отделяющие путь от имени	SysUtils.hpp
ExtractRelativePath	System::AnsiString ExtractRelativePath(const System::AnsiString BaseName, const System::AnsiString DestName) Возвращает относительный путь файла DestName относительно каталога BaseName , включая форматы вида «..\»	SysUtils.hpp
ExtractShortPathName	System::AnsiString ExtractShortPathName(const System::AnsiString FileName) Возвращает путь и имя файла FileName , преобразовывая имена в формат 8.3	SysUtils.hpp

Функция	Синтаксис / Описание	Файл
MatchesMask	<pre>bool MatchesMask(const System::AnsiString Filename, const System::AnsiString Mask)</pre> <p>Проверяет Filename на использование маски Mask</p>	Masks.hpp
MinimizeName	<pre>System::AnsiString MinimizeName(const System::AnsiString Filename, Graphics::TCanvas * Canvas, int MaxLen)</pre> <p>Минимизирует имя Filename, сокращая путь до размера, вмещающегося при изображении на канве Canvas в MaxLen пикселей</p>	filectrl.hpp
ProcessPath	<pre>void ProcessPath(const System::AnsiString EditText, char &Drive, System::AnsiString &DirPart, System::AnsiString &FilePart)</pre> <p>Разделяет путь процесса EditText на драйвер Drive, путь DirPart и имя FilePart</p>	filectrl.hpp
tmpnam	<pre>char *tmpnam(char *s)</pre> <p>Создает уникальное имя файла, которое может в дальнейшем использоваться для создания временных файлов</p>	stdio.h

Комментарии

Все функции (кроме **tmpnam** и **_mktemp**) возвращают строку типа **AnsiString** (см. главу 16), содержащую результат преобразования имени файла **FileName**. Могут работать с многобайтными символами. Если **FileName** не содержит пути или расширения, то соответствующие функции **ExtractFile...** возвращают пустую строку.

Не забывайте, что обратный слеш в строке пути к файлу должен повторяться дважды (см. раздел 15.1.3). Например:

```
"C:\\Program Files\\Bcb.exe"
```

Функция **ExtractFileDir** возвращает путь к файлу в том виде, который нужен для передачи в функции **CreateDir**, **GetCurrentDir**, **RemoveDir**, **SetCurrentDir** (см. раздел 15.5.6).

Функция **ExtractShortPathName** возвращает путь и имя файла **FileName**, сокращая имена каталогов и файлов до формата 8.3. Например, строка

```
C:\Program Files\Borland\CBuilder\Bin\Bcb.exe
```

будет возвращена как

```
C:\Progra~1\Borland\CBuilder\Bin\Bcb.exe
```

Функция **MatchesMask** проверяет **Filename** на использование маски **Mask**. Маска может содержать обычные алфавитно-цифровые символы, множества, символы «*» — любое количество любых символов и «?» — любой один символ. Множества заключаются в квадратные скобки []. В скобках без пробелов записываются возможные символы или диапазоны в виде <символ>-<символ>. Например, «a-c». Если первый символ в множестве — восклицательный знак «!», то это множество содержит символы, которые не должны встречаться. Сравнение с маской производится без учета регистра. Функция **MatchesMask** возвращает **true**, если **Filename** соответствует маске, **false**, если не соответствует и генерирует исключение при синтаксически неверной маске. Например, маске

```
"[a-b]:\\test\\*.*"
```

будут соответствовать все имена файлов, расположенных на дисковых **a:** или **b:** в каталоге «test».

Функция **MinimizeName** минимизирует имя **Filename**, сокращая путь до размера, вмещающегося при изображении на канве **Canvas** в **MaxLen** пикселей. Вместо выброшенных частей пути изображаются точки. Например, оператор

```
Label1->Caption = MinimizeName("C:\\Program Files\\Bcb.exe",
                                Label1->Canvas, 100);
```

приведет к появлению в метке **Label1** надписи:

```
C:\...\Bcb.exe
```

Функции **tmpnam** и **_mktemp** создают уникальное имя файла, которое может в дальнейшем использоваться для создания временных файлов. Последовательное обращение к функции **tmpnam** может создавать до **TMP_MAX** = 65 535 уникальных имен. Параметр **s** этой функции должен задаваться или равным **NULL**, или указывать на массив размером не менее **L_tmpnam** символов (эта константа определена в **stdio.h**). В случае параметра равного **NULL**, функция **tmpnam** сама создает необходимый объект с именем файла и возвращает указатель на него.

Если вы создаете затем временный файл с именем, сгенерированным **tmpnam** или **_mktemp**, то должны сами позаботиться о его удалении в конце работы программы. Автоматически он не удаляется.

15.5.6 Управление каталогами и файлами на дисках

Функция	Синтаксис / Описание	Файл
_getcwd	char * _getcwd(int drive, char *buffer, int buflen) Заносит в буфер buffer размером buflen текущий каталог диска drive (0 — текущий диск, 1 — A и т.д.); возвращает указатель на buffer или NULL ; при buffer = NULL создает буфер и возвращает указатель на него	direct.h
_rmdir	int _rmdir(const char *path) Удаляет каталог path (пустой, не текущий и не корневой); возвращает 0 при успехе или -1	dir.h
_rtl_chmod	int _rtl_chmod(const char *path, int func [, int attrib]) При func = 0 возвращает текущие атрибуты файла, при func = 1 устанавливает файлу атрибуты attrib	io.h, dos.h
_unlink	int _unlink(const char *filename) Удаляет с диска файл filename ; возвращает 0 или -1	io.h
_waccess	int _waccess(const wchar_t *filename, int amode) Определяет, существует ли файл filename и какие операции с ним доступны; режим работы задается параметром amode	io.h
_wrtl_chmod	int _wrtl_chmod(const wchar_t *path, int func, ...) При func = 0 возвращает текущие атрибуты файла, при func = 1 устанавливает файлу атрибуты attrib	io.h, dos.h

Функция	Синтаксис / Описание	Файл
access	int access(const char *filename, int amode) Определяет, существует ли файл filename и какие операции с ним доступны; режим работы задается параметром amode	io.h
chdir	int chdir(const char *path) Задаёт path в качестве текущего каталога; возвращает 0 при успехе или -1	dir.h
chmod	int chmod(const char *path, int amode) Изменяет режим доступа amode к файлу path ; возвращает 0 при успехе или -1; amode содержит одно или оба значения S_IWRITE и S_IREAD	io.h
chsize	int chsize(int handle, long size) Изменяет размер файла с дескриптором handle , открытого для записи, до size байтов; возвращает 0 или -1	io.h
CreateDir	bool CreateDir(const System::AnsiString Dir) Создаёт каталог Dir и возвращает true в случае успеха	Sys-Utils.hpp
DeleteFile	bool DeleteFile(const System::AnsiString FileName) Удаляет файл FileName с диска и возвращает true в случае успеха	Sys-Utils.hpp
DirectoryExists	bool DirectoryExists(const System::AnsiString Name) Определяет, существует ли каталог Name	Sys-Utils.hpp
DiskFree	int DiskFree(Byte Drive) Возвращает число свободных байтов на диске Drive или -1, если Drive ошибочный (Drive = 0 — текущий диск, 1 — A, 2 - B и т.д.)	Sys-Utils.hpp
DiskSize	int DiskSize(Byte Drive) Возвращает размер в байтах диска Drive или -1, если Drive ошибочный (Drive = 0 — текущий диск, 1 — A, 2 - B и т.д.)	Sys-Utils.hpp
FileAge	int FileAge(const System::AnsiString FileName) Возвращает дату создания файла FileName или -1, если такого файла нет	Sys-Utils.hpp
FileDateTo-DateTime	System::TDateTime FileDateToDateTime(int FileDate) Возвращает в формате типа TDateTime дату и время FileDate , заданные в формате DOS	Sys-Utils.hpp
FileExists	bool FileExists(const System::AnsiString FileName) Определяет, существует ли файл FileName	Sys-Utils.hpp
FileGetAttr	int FileGetAttr(const System::AnsiString FileName) Возвращает атрибуты файла FileName	Sys-Utils.hpp

Функция	Синтаксис / Описание	Файл
FileGetDate	int FileGetDate(int Handle) Возвращает дату создания файла с дескриптором Handle или -1, если такого файла нет	Sys-Utills.hpp
filelength	long filelength(int handle) Возвращает длину в байтах файла с дескриптором handle ; при ошибке возвращает -1	io.h
FileSearch	System::AnsiString FileSearch(const System::AnsiString Name, const System::AnsiString DirList) Ищет в списке каталогов DirList файл Name ; возвращает полный путь к файлу или пустую строку	Sys-Utills.hpp
FileSetAttr	int FileSetAttr(const System::AnsiString FileName, int Attr) Устанавливает файлу FileName атрибуты Attr ; возвращает 0 или код ошибки	Sys-Utills.hpp
FileSetDate	int FileSetDate(int Handle, int Age) Устанавливает дату Age файлу с дескриптором Handle ; возвращает 0 или код ошибки	Sys-Utills.hpp
FindClose	void FindClose(TSearchRec &F) Завершает последовательность поиска функциями FindFirst и FindNext со структурой F и освобождает память	Sys-Utills.hpp
FindFirst	int FindFirst(const System::AnsiString Path, int Attr, TSearchRec &F) Начинает поиск файлов по шаблону Path с атрибутами Attr ; заносит результат в F ; возвращает 0 или код ошибки	Sys-Utills.hpp
findfirst	int findfirst(const char _FAR * __path, struct ffbk _FAR * __ffbk, int __attrib) Начинает поиск файлов по шаблону __path с атрибутами __attrib ; заносит результат в __ffbk ; возвращает 0 при успехе или -1	dir.h
FindNext	int FindNext(TSearchRec &F) Продолжает поиск файлов, начатый функцией FindFirst со структурой F ; заносит результат в F ; возвращает 0 или код ошибки	Sys-Utills.hpp
findnext	int findnext(struct ffbk _FAR * __ffbk) Продолжает поиск файлов, начатый функцией findfirst со структурой __ffbk ; возвращает 0 при успехе или -1	dir.h
fnmerge	void fnmerge(char *path, const char *drive, const char *dir, const char *name, const char *ext) Формирует строку path пути к файлу из его отдельных составляющих: диска drive , каталога dir , имени файла name и расширения ext	dir.h

Функция	Синтаксис / Описание	Файл
fnsplit	int fnsplit(const char *path, char *drive, char *dir, char *name, char *ext) Разделяет строку path пути к файлу на его отдельные составляющие: диск drive , каталог dir , имя файла name и расширение ext	dir.h
ForceDirectories	void ForceDirectories(System::AnsiString Dir) Создает каталог Dir и все промежуточные родительские каталоги, если они отсутствуют	File-Ctrl.hpp
fstat	int fstat(int handle, struct stat *statbuf) Заносит в структуру statbuf информацию об открытом файле с дескриптором handle ; возвращает 0 или -1	sys\stat.h
getcurdir	int getcurdir(int drive, char *directory) Заносит в directory текущий каталог диска drive (0 — текущий диск, 1 — A и т.д.) без имени диска и начального символа «\»	dir.h
GetCurrentDir	System::AnsiString GetCurrentDir() Возвращает текущий каталог	Sys-Utils.hpp
getcwd	char *getcwd(char *buf, int buflen) Возвращает и сохраняет в буфере buf размером buflen полный путь к текущему каталогу, включая диск; возвращает указатель на buf или NULL ; при buf = NULL создает буфер и возвращает указатель на него	dir.h
getdisk	int getdisk(void) Возвращает текущий диск: 0 — A, 1 — B и т.д.	dir.h
getftime	int getftime(int handle, struct ftime *ftimep) Читает время и дату создания файла handle в структуру ftimep ; возвращает 0 или -1	io.h
GetSystemDirectory	UINT GetSystemDirectory(LPTSTR lpBuffer, UINT uSize) Функция API Windows, заносит в буфер lpBuffer размером uSize системный каталог Windows	-
GetWindowsDirectory	UINT GetWindowsDirectory(LPTSTR lpBuffer, UINT uSize) Функция API Windows, заносит в буфер lpBuffer размером uSize каталог Windows	-
isatty	int isatty(int handle) Возвращает ненулевое значение, если файл с дескриптором handle связан с одним из устройств: терминал, консоль, принтер, последовательный порт	io.h
mkdir	int mkdir(const char *path) Создает каталог path ; возвращает 0 при успехе или -1	dir.h

Функция	Синтаксис / Описание	Файл
remove	int remove(const char *filename) Макрос, удаляет с диска файл filename ; возвращает 0 или -1	stdio.h
RemoveDir	bool RemoveDir(const System::AnsiString Dir) Удаляет с диска каталог Dir	Sys-Utils.hpp
rename	int rename(const char *oldname, const char *newname) Переименовывает файл oldname , давая ему новое имя newname ; может использоваться для перемещения файла без изменения диска; возвращает 0 или -1	stdio.h
RenameFile	bool RenameFile(const System::AnsiString OldName, const System::AnsiString NewName) Переименовывает файл OldName , давая ему новое имя NewName ; если файл с именем NewName уже существует или нет файла OldName , возвращается false	Sys-Utils.hpp
searchpath	char *searchpath(const char *file) Ищет файл file в каталогах, указанных в переменной окружения PATH ; возвращает полный путь к файлу или NULL	dir.h
SetCurrent-Dir	bool SetCurrentDir(const System::AnsiString Dir) Задаёт Dir в качестве текущего каталога	Sys-Utils.hpp
setdisk	int setdisk(int drive) Устанавливает в качестве текущего диск drive : 0 — A, 1 — B и т.д.; возвращает число доступных дисков	dir.h
setftime	int setftime(int handle, struct ftime *ftimep) Устанавливает время и дату создания файла handle по данным структуры ftimep ; возвращает 0 или -1	io.h
stat	int stat(const char *path, struct stat *statbuf) Заносит в структуру statbuf информацию об открытом файле path ; возвращает 0 или -1	sys\stat.h

Комментарии

Функции **fstat** и **stat** заносят в структуру типа **stat** информацию об открытом файле. Структура имеет поля:

st_mode	битовая маска режима файла
st_dev	номер диска файла или дескриптор, если файл на устройстве
st_rdev	то же, что st_dev
st_nlink	константа 1
st_size	размер файла в байтах
st_atime	время последнего открытия (в Windows) или изменения (в DOS)
st_mtime	то же, что st_atime
st_ctime	то же, что st_atime

Маска **st_mode** содержит информацию о режиме открытого файла и включает в себя следующие биты.

Должен быть установлен один из следующих битов:

S_IFCHR	если дескриптор ссылается на устройство
S_IFREG	если дескриптор ссылается на обычный файл

Должен быть установлен один или оба следующих битов:

S_IWRITE	пользователю разрешена запись в файл
S_IREAD	пользователю разрешено чтение из файла

Системы HPFS и NTFS учитывают следующее различие между полями **st_atime**, **st_mtime** и **st_ctime**:

st_atime	время последнего доступа к файлу
st_mtime	время последней модификации файла
st_ctime	время создания файла

Следующий пример демонстрирует работу функции **stat**:

```
#include <stdio.h>
#include <time.h>
#include <sys\stat.h>
...
struct stat statbuf;
FILE *stream;
if ((stream = fopen("TEST.TXT", "r+t")) == NULL)
{
    ShowMessage("Невозможно открыть файл");
    return;
}
stat("TEST.TXT", &statbuf);           // чтение информации
fclose(stream);

// отображение информации:
if (statbuf.st_mode & S_IFCHR)
    Memol->Lines->Add("Дескриптор устройства");
if (statbuf.st_mode & S_IFREG)
    Memol->Lines->Add("Ссылка на файл");
if (statbuf.st_mode & S_IREAD)
    Memol->Lines->Add("Разрешено чтение из файла");
if (statbuf.st_mode & S_IWRITE)
    Memol->Lines->Add("Разрешена запись в файл");
Memol->Lines->Add("Файл расположен на диске " +
    AnsiString((char)('A'+statbuf.st_dev)));
Memol->Lines->Add("Размер файла в байтах: " +
    IntToStr(statbuf.st_size));
Memol->Lines->Add("Последний раз файл был открыт " +
    AnsiString(ctime(&statbuf.st_ctime)));
```

Функции **remove** и **_unlink** удаляют с диска указанный файл. Если файл открыт, то перед удалением его надо закрыть. Файл с атрибутом только для чтения не может быть удален. Сначала надо изменить его атрибут функциями **chmod** или **_rtl_chmod**.

Функции **access** и **_waccess** определяют разрешенный доступ к файлу, заданному параметром **filename**. Различаются они только типом строки, содержащей имя файла. Функции проверяют, существует ли файл, и если существует, то разрешено ли чтение, запись или выполнение этого файла.

Параметр **amode** определяет, что именно должно проверяться:

06	Проверка разрешения чтения и записи
04	Проверка разрешения чтения
02	Проверка разрешения записи
01	Проверка, является ли файл выполняемым
00	Проверка существования файла

В DOS, OS/2 и Windows все существующие файлы имеют разрешение чтения. Поэтому значения **amode** 00 и 04 дают одинаковый результат. В DOS разрешение записи подразумевает и разрешение чтения. Поэтому 06 и 02 также дают одинаковый результат.

Если в качестве **filename** задан не файл, а каталог, то функции просто проверяют, существует ли каталог.

Функции возвращают 0, если файл имеет запрошенный уровень доступа. В противном случае возвращается 1, а глобальная переменная **errno** устанавливается в одно из следующих состояний:

- ENOENT** путь или файл не найден
- EACCES** ошибка доступа

Функции **FindFirst**, **FindNext** и **FindClose** осуществляют поиск файлов, удовлетворяющих некоторым условиям. В своей работе они используют для хранения информации о файле структуру **TSearchRec**:

```
struct TSearchRec
{
    int Time;                // время создания
    int Size;                // размер файла
    int Attr;                // атрибуты
    System::AnsiString Name; // имя файла
    int ExcludeAttr;
    int FindHandle;          // дескриптор
    _WIN32_FIND_DATA FindData;
};
```

Поле **Attr** этой структуры содержит слово атрибутов файла. Атрибуты файлов определяются комбинациями следующих именованных констант:

Имя константы	Значение	Атрибут
faReadOnly	\$00000001	файл только для чтения
faHidden	\$00000002	невидимый файл
faSysFile	\$00000004	системный файл
faVolumeID	\$00000008	идентификатор диска
faDirectory	\$00000010	каталог
faArchive	\$00000020	архивный файл
faAnyFile	\$0000003F	любой файл

Начинается поиск вызовом функции **FindFirst**. В нее в качестве аргумента **Path** передается шаблон поиска, а в качестве аргумента **Attr** — слово атрибутов, которыми должны характеризоваться искомые файлы. Это слово формируется из указанных выше констант атрибутов операцией поразрядного ИЛИ. Если соответствующий файл найден, функция возвращает 0, а в указанную ее параметром **F** структуру типа **TSearchRec** заносится информация о файле. Если после этого вызвать функцию **FindNext** с той же структурой **F** в качестве параметра, то будет продолжен поиск следующего файла, удовлетворяющего заданным условиям. После того, как поиск всех файлов закончен или после того, как решено поиск прервать, вызывается функция **FindClose**, которая освобождает выделенную под организацию поиска память.

Например, следующие операторы осуществляют поиск всех файлов и подкаталогов текущего каталога и выводят результаты в окно редактирования **Memol**:

```
TSearchRec sr;
Memol->Clear();
if (FindFirst("*.\"", faAnyFile | faDirectory, sr) == 0)
{
    Memol->Lines->Add(sr.Name+", размер: "+IntToStr(sr.Size));
    while (FindNext(sr) == 0)
        Memol->Lines->Add(sr.Name+", размер: "+IntToStr(sr.Size));
}
FindClose(sr);
```

Приведем более сложный пример. Следующий обработчик щелчка на кнопке **Button1** обеспечивает в текущем каталоге и во всех его подкаталогах поиск и удаление файлов с расширением **.tds**:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    AnsiString CurDir = GetCurrentDir();
    TSearchRec sr;
    if (FindFirst("*.\"", faAnyFile | faDirectory, sr) == 0)
    {
        if (CompareText(ExtractFileExt(sr.Name), ".tds") == 0)
            DeleteFile(sr.Name);
        else if ((sr.Attr == faDirectory) && (sr.Name != ".")
            && (sr.Name != ".."))
        {
            SetCurrentDir(ExtractFileDir(sr.Name));
            Button1Click(Sender);
            SetCurrentDir(CurDir);
        }
        while (FindNext(sr) == 0)
        {
            if (CompareText(ExtractFileExt(sr.Name), ".tds") == 0)
                DeleteFile(sr.Name);
            else if ((sr.Attr == faDirectory) && (sr.Name != ".")
                && (sr.Name != ".."))
            {
                SetCurrentDir(sr.Name);
                Button1Click(Sender);
                SetCurrentDir(CurDir);
            }
        }
    }
    FindClose(sr);
}
```

Приведенная функция использует рекурсивный вызов самой себя при переходе в подкаталоги.

Альтернативный вариант поиска файлов, удовлетворяющих шаблону, предоставляют функции **findfirst** и **findnext**. Они используют для хранения информации о файле структуру типа **ffblk**:

```
struct ffbk {
    long          ff_reserved;
    long          ff_fsize;      // размер файла
    unsigned long ff_attrib;     // атрибуты
    unsigned short ff_ftime;     // время создания
    unsigned short ff_fdate;     // дата создания
    char          ff_name[256];  // имя файла
};
```

Поле атрибутов **ff_attrib** в этой структуре формируется из констант **FA_...**, определенных в файле **dos.h** и рассмотренных в разделе 15.5.1. Поля **ff_ftime** и **ff_fdate** содержат битовые поля даты и времени. В поле **ff_ftime** биты 0 - 4 хранят пары секунд, биты 5 - 10 хранят минуты, биты 11 - 15 — часы. В поле **ff_fdate** биты 0 - 4 хранят день, биты 5 - 8 — месяц, биты 9 - 15 — год, отсчитываемый от 1980.

Сам по себе поиск, осуществляемый функциями **findfirst** и **findnext**, не отличается от рассмотренного выше для функций **FindFirst** и **FindNext**.

Функция **FileSearch** ищет в списке каталогов **DirList** файл **Name**. Список **DirList** представляет собой перечень обычным образом записанных путей к каталогам, разделенных точками с запятой. При успешном поиске функция возвращает имя файла с полным путем к нему (если файл найден в текущем каталоге, возвращается только имя файла). Если файл не найден, возвращается пустая строка.

Функция **searchpath** ищет файл **file** в списке каталогов, указанных в переменной окружения **PATH**. Возвращает полный путь к файлу или **NULL**.

Функция **FileExists** определяет, существует ли указанный файл.

Функция **fnmerge** формирует строку **path** пути к файлу из его отдельных составляющих: диска **drive**, каталога **dir**, имени файла **name** и расширения **ext**. В итоге получается строка вида **drive:\path\name.ext**. Каждая из составляющих пути (**dir**, **path**, **name**, **ext**) может быть задана равной **NULL**. Тогда эта составляющая в результирующий путь не включается. Функция **fnsplit** осуществляет обратную операцию: разделяет полный путь к файлу на его составляющие.

Функция **DirectoryExists** определяет, существует ли каталог **Name**. Если **Name** содержит полный путь, то проверяется наличие именно указанного каталога. В противном случае **Name** воспринимается как путь относительно текущего каталога.

Функции **CreateDir** и **ForceDirectories** создают переданный им как параметр каталог **Dir**. Функция **ForceDirectories** отличается от других подобных функций тем, что она может создать не только конечный каталог, но одновременно и его родительские каталоги, если они отсутствуют. Например, оператор

```
ForceDirectories("C:\\Test\\Test1");
```

создаст не только каталог **Test1**, но и его родительский каталог **Test**, если он отсутствует. Проверить, создан ли нужный каталог этой функцией, можно с помощью функции **DirectoryExists**.

Функции **getcurdir** и **GetCurrentDir** позволяют определить текущий каталог на заданном или текущем диске.

Функция API Windows **GetSystemDirectory**, заносит в буфер строку, характеризующую системный каталог Windows. Это тот каталог, в котором размещены файлы библиотек, драйверов, шрифтов. Приложение не должно создавать какие-то файлы в системном каталоге. Создавать свои файлы можно в каталоге, возвращаемом другой аналогичной функцией — **GetWindowsDirectory**, дающей путь к каталогу Windows. Этот каталог содержит файлы приложений Windows, файлы инициализации **.ini** и файлы справок **.hlp**. В этом каталоге вы можете хранить

файлы инициализации и файлы справок своего приложения. Если приложение создает другие файлы, которые вы хотите хранить, не допуская к ним других пользователей, то помещайте их в каталог, указанный в переменной окружения **НОМЕРАТН**. При соответствующей установке этот каталог различен для всех пользователей.

Параметр **lpBuffer** функций **GetSystemDirectory** и **GetWindowsDirectory** является указателем на строку с нулевым символом в конце, в которую передается найденный путь. Этот путь записывается без заключительного обратного слеша «\», если только каталог не является корневым.

Параметр **uSize** указывает максимальный размер буфера в символах. Его величина должна быть не менее значения **MAX_PATH**.

При успешном выполнении функции копируют путь в **lpBuffer** и возвращают число символов в строке, не считая последнего нулевого. Если длина строки больше, чем **uSize**, то возвращенное значение позволяет узнать требуемый размер буфера.

Если функция не смогла успешно завершиться, то она возвращает нулевое значение. В этом случае узнать причину отказа можно, вызвав **GetLastError** (см. раздел 15.7.5).

Приведем пример. Если ваш системный каталог Windows назван **WINDOWS\SYSTEM** и расположен на диске C:, то операторы

```
char s[MAX_PATH];
GetSystemDirectory(s, MAX_PATH);
```

занесут в **s** путь: **C:\WINDOWS\SYSTEM**. Полученный путь можно, например, использовать для проверки, имеются ли на компьютере пользователя нужные библиотеки, драйверы и шрифты.

Функции **FileGetAttr**, **FileSetAttr**, **_rtl_chmod** позволяют определять или устанавливать атрибуты файла (см. описание атрибутов в разделе 15.5.1). Устанавливая атрибуты их можно объединять в одно слово атрибутов операцией поразрядного ИЛИ. Возвращенные функциями **FileGetAttr** и **_rtl_chmod** атрибуты можно проверять с помощью операции поразрядного И.

Функция **_rtl_chmod** позволяет определить или установить атрибуты файла (см. их описание в разделе 15.5.1). При **func = 0** функция возвращает слово текущих атрибутов файла **path**, а при **func = 1** устанавливает файлу **path** атрибуты **attrib**. Например, следующий оператор устанавливает для файла, заданного строкой **SFile**, атрибут «невидимый»:

```
_rtl_chmod(SFile, 1, FA_HIDDEN);
```

Возвращенные функциями **FileGetAttr** и **_rtl_chmod** атрибуты можно проверять с помощью операции поразрядного И.

Следующие два оператора добавляют к атрибутам файла, заданного строкой **SFile**, атрибут «невидимый»:

```
int attrib = _rtl_chmod(SFile, 0);
_rtl_chmod(SFile, 1, attrib | FA_HIDDEN);
```

Следующие операторы определяют и отображают в окне редактирования **Mem1** атрибуты файла **SFile**:

```
int attrib = _rtl_chmod(SFile, 0);
if (attrib == -1)
{
    Mem1->Lines->Add("SrP<LB RTXDx " + IntToStr(errno));
    return;
}
if (attrib & FA_RDONLY)
    Mem1->Lines->Add(SFile + " - файл только для чтения");
if (attrib & FA_HIDDEN)
    Mem1->Lines->Add(SFile + " - невидимый файл");
```

```

if (attrib & FA_SYSTEM)
    Mem01->Lines->Add(SFile + " - системный файл");
if (attrib & FA_DIRECT)
    Mem01->Lines->Add(SFile + " - каталог");
if (attrib & FA_ARCHIVE)
    Mem01->Lines->Add(SFile + " - архивный файл");

```

Функции **FileAge**, **FileGetDate** и **FileSetDate** оперируют с данными о времени создания файла в формате DOS. В этом же формате хранится значение поля **Time** структуры типа **TSearchRec**, используемой в функциях **FindFirst** и **FindNext**. Преобразование этого формата в тип **TDateTime** может осуществляться функцией **FileDateToDateTime**. См. также раздел 15.3.2, посвященный преобразованиям форматов дат и времени.

Функция **getftime** читает время и дату создания файл, заданного своим дескриптором **handle** (он может быть определен функцией **fileno**), и заносит их в структуру типа **ftime**, на которую указывает параметр **ftimep**. Функция **setftime** выполняет обратную задачу: задает файлу время и дату в соответствии с данными, записанными в структуру **ftimep**. Файл должен быть доступен для записи. В противном случае произойдет ошибка **EACCES**. После установки времени и даты в файл ничего нельзя записывать, пока он не закрыт. Иначе установка будет изменена.

При успешном завершении функции возвращают 0. В противном случае возвращается -1, а глобальная переменная **errno** устанавливается в одно из следующих состояний:

EACCES ошибка доступа
EBADF ошибочный номер файла
EINVFNC ошибочный номер функции

Структура типа **ftime** имеет вид:

```

struct ftime {
    unsigned ft_tsec: 5;           // пары секунд
    unsigned ft_min: 6;           // минуты
    unsigned ft_hour: 5;          // часы
    unsigned ft_day: 5;           // день
    unsigned ft_month: 4;         // месяц
    unsigned ft_year: 7;          // год - 1980
};

```

Следующий код в качестве примера определяет дату создания файла «Test.txt», уменьшает день на 1 (делает дату вчерашней — предполагается, что день не равен 1) и устанавливает файлу эту измененную дату:

```

FILE *stream;
std::ftime ft;
char buffer[80];
if ((stream = fopen("TEST.TXT", "r+t")) == NULL)
{
    ShowMessage("Невозможно открыть файл для записи");
    return;
}
getftime(fileno(stream), &ft);           // чтение даты
sprintf(buffer, "Дата создания файла: %u/%u/%u",
        ft.ft_day, ft.ft_month, ft.ft_year+1980);
ShowMessage(buffer);
ft.ft_day--;                             // изменение даты
setftime(fileno(stream), &ft);           // установка даты
fclose(stream);

```

15.6 Управление процессами

15.6.1 Функции управления текущим процессом

Функция	Синтаксис / Описание	Файл
_c_exit	void _c_exit(void) Выполняет все действия, аналогичные функции exit , по закрытию файлов и очистке буферов, но не вызывает функций окончания и не прерывает выполнение программы	process.h
_cexit	void _cexit(void) Выполняет все действия, аналогичные функции exit , по закрытию файлов, очистке буферов и вызову функций окончания, но не прерывает выполнение программы	process.h
_exit	void _exit(int status) Завершает выполнение программы, но в отличие от exit не сбрасывает буферы, не закрывает файлы и не вызывает функции окончания; status — устанавливаемый код завершения	stdlib.h
abort	void abort(void) Аварийное завершение программы	stdlib.h
atexit	int atexit(void (_USERENTRY * func)(void)) Регистрирует функцию окончания func ; при успехе возвращает 0	stdlib.h
exit	void exit(int status) Завершает выполнение программы, закрывая все открытые файлы, сбрасывая выходные буферы в соответствующие потоки, и вызывая все зарегистрированные функцией atexit функции окончания; status — устанавливаемый код завершения	stdlib.h
raise	int raise(int sig) Генерирует сигнал sig ; при успешной генерации возвращает 0	signal.h
signal	void (_USERENTRY *signal(int sig, void (_USERENTRY *func) (int sig[, int subcode])))(int) Задание обработчика func сигнала sig	signal.h

Комментарии

Функция **atexit** регистрирует в системе функцию окончания. Эта функция будет вызываться при завершении работы программы функциями **exit** и **_cexit**. Обычно в этой функции предусматривается «зачистка мусора» — освобождение динамически распределенной памяти, уничтожение временных файлов, разрыв соединений с базами данных и т.п. Например, оператор

```
atexit(Exit1);
```

регистрирует функцию окончания, которую вы можете записать в виде:


```
void Exit1(void)
{
    ...
}
```

Всего в приложении может быть зарегистрировано до 32 функций окончания. При завершении приложения они срабатывают в последовательности, обратной последовательности их регистрации, т.е. последняя зарегистрированная функция будет вызываться первой.

Семейство функций **exit** выполняет операции по завершению работы приложения. Наиболее полное завершение выполняет функция **exit**. Прежде, чем завершить приложение, она закрывает все открытые файлы, сбрасывает выходные буферы в соответствующие потоки, вызывает все зарегистрированные функцией **atexit** функции окончания. Параметр **status** функции **exit** — это устанавливаемый код завершения.

Приведенная ниже таблица показывает, какие из этих операций выполняются другими функциями семейства **exit**, а какие нет.

Функция	Заккрытие файлов	Сброс буферов	Вызов функций окончания	Завершение приложения
exit	+	+	+	+
_exit	-	-	-	+
_cexit	+	+	+	-
_c_exit	+	+	-	-

Функция **signal** указывает обработчик **func** сигнала, переданного в нее параметром **sig**. Сигнал — это некоторое непредвиденное событие (прерывание), которое может вызвать преждевременное завершение программы. Его основное отличие от исключения в том, что после генерации сигнала можно вернуться в ту точку кода, в которой был сгенерирован сигнал. В случае исключения это невозможно. Сигналы могут поступать от внешних устройств, генерироваться в результате некоторых аварийных ситуаций или преднамеренно генерироваться функцией **raise**.

В файле **signal.h** предопределены следующие сигналы:

SIGABRT	Аварийное завершение программы. Генерируется только вызовом функций abort , raise и необработанными исключениями. Действие по умолчанию — вызов _exit(3)
SIGBREAK	Прерывание нажатием клавиш Ctrl-Break
SIGFPE	Ошибка арифметической операции, например, деления на нуль или операции, вызвавшей переполнение. Действие по умолчанию — вызов _exit(1)
SIGILL	Появление в коде недопустимой команды. Действие по умолчанию — вызов _exit(1)
SIGINT	Получение интерактивного сигнала (например, прерывание Ctrl+C). Действие по умолчанию — прерывание INT 23h
SIGSEGV	Нарушение доступа к памяти. Действие по умолчанию — вызов _exit(1)
SIGTERM	«Мягкое» завершение процесса. Действие по умолчанию — вызов _exit(1)

SIGUSR1,	Определенные пользователем (только в Win32) сигналы пользователя, генерируемые функцией raise . Действие по умолчанию — игнорирование сигнала
SIGUSR2,	
SIGUSR3	

В функцию **signal** передаются два параметра: целочисленный номер сигнала **sig** и указатель **func** на функцию обработки сигнала. Обработчик **func** может быть определенной пользователем функцией или одним из трех предопределенных обработчиков: **SIG_DFL** — завершение программы, **SIG_IGN** — игнорирование сигналов данного вида, и **SIG_ERR** — генерация ошибки. Например:

```
signal(SIGINT, SIG_ERR); // генерация сообщения об ошибке
```

или

```
signal(SIGINT, SIG_IGN); // игнорирование сигнала SIGINT
```

Если вы хотите задать свой обработчик стандартного сигнала или определенного вами сигнала, это может выглядеть так. Пусть, например, вы хотите предусмотреть в своей программе обработчик некоторого вводимого вами сигнала **SIGUSR1**. Назовем функцию этого обработчика **Handl_SIGUSR1**. Определите в программе тип указателя на функцию **fptr**:

```
typedef void (*fptr)(int);
```

Не забудьте также вставить в файл директиву

```
#include <signal.h>
```

Далее где-то в начале программы (например, в событии формы **OnCreate**) надо ввести оператор:

```
signal(SIGUSR1, (fptr)Handl_SIGUSR1);
```

Этот оператор установит функцию **Handl_SIGUSR1** как обработчик сигнала **SIGUSR1**. В нужных местах программы вставьте оператор генерации вашего сигнала:

```
raise(SIGUSR1);
```

Рассмотрим подробнее последовательность операций системы при выполнении этого оператора. Если в программе задан приведенным выше оператором **signal** обработчик пользователя для данного события, то происходит обращение к этому обработчику, но прежде система сбрасывает установку на **SIG_DFL**. Это значит, что при следующей генерации этого сигнала, если не повторить вызов функции **signal**, система забудет прежнюю установку и не будет опять обращаться к обработчику пользователя. Поэтому обычно в конце обработчика повторяют вызов **signal**. С учетом этого обработчик сигнала может иметь вид:

```
void Handl_SIGUSR1(int N)
{
    ...
    if(MessageDlg("Продолжать?", mtConfirmation,
                  TMsgDlgButtons() << mbYes << mbNo, 0) == mrYes)
        // повторная установка обработчика:
        signal(SIGUSR1, Handl_SIGUSR1);

    else exit(EXIT_SUCCESS);
}
```

Обработчик принимает одно целое значение, соответствующее номеру сигнала. В нем предусматриваются некоторые действия, необходимые при появлении данного сигнала. Затем, если выполнение программы должно продолжаться, надо повторно установить обработчик сигнала с помощью функции **signal**, как показано в приведенном примере.

Функция **abort** вызывает аварийное завершение программы генерацией сигнала **SIGABRT**. Если в приложении не предусмотрен обработчик этого сигнала, то функция **abort** заносит сообщение «Abnormal program termination» в поток ошибок **stderr** и завершает выполнение программы вызовом **_exit** с кодом завершения 3.

15.6.2 Функции выполнения порождаемых процессов exec... и spawn...

Функция	Синтаксис / Описание	Файл
cwait	int cwait(int *statloc, int pid, int action); Обеспечивает ожидание завершения указанного порожденного процесса, заносит в statloc статус завершения, возвращает ID порожденного процесса или -1	process.h
execl	int execl(char *path, char *arg0, *arg1, ..., *argn, NULL) Выполняет порожденный процесс path с аргументами arg0 - argn	process.h
execle	int execle(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env) Выполняет порожденный процесс path с аргументами arg0 - argn и с окружением env	process.h
execlp	int execlp(char *path, char *arg0, *arg1, ..., *argn, NULL) Выполняет порожденный процесс path с аргументами arg0 - argn , с поиском в PATH	process.h
execlpe	int execlpe(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env) Выполняет порожденный процесс path с аргументами arg0 - argn , с поиском в PATH и с окружением env	process.h
execv	int execv(char *path, char *argv[]) Выполняет порожденный процесс path с аргументами argv[]	process.h
execve	int execve(char *path, char *argv[], char **env) Выполняет порожденный процесс path с аргументами argv[] и с окружением env	process.h
execvp	int execvp(char *path, char *argv[]) Выполняет порожденный процесс path с аргументами argv[] , с поиском в PATH	process.h
execvpe	int execvpe(char *path, char *argv[], char **env) Выполняет порожденный процесс path с аргументами argv[] , с поиском в PATH и с окружением env	process.h
spawnl	int spawnl(int mode, char *path, char *arg0, arg1, ..., argn, NULL) Выполняет в режиме mode порожденный процесс path с аргументами arg0 - argn	process.h, stdio.h

Функция	Синтаксис / Описание	Файл
spawnle	int spawnle(int mode, char *path, char *arg0, arg1, ..., argn, NULL, char *envp[]) Выполняет в режиме mode порожденный процесс path с аргументами arg0 - argn и с окружением envp	process.h, stdio.h
spawnlp	int spawnlp(int mode, char *path, char *arg0, arg1, ..., argn, NULL) Выполняет в режиме mode порожденный процесс path с аргументами arg0 - argn , с поиском в PATH	process.h, stdio.h
spawnlpe	int spawnlpe(int mode, char *path, char *arg0, arg1, ..., argn, NULL, char *envp[]) Выполняет в режиме mode порожденный процесс path с аргументами arg0 - argn , с поиском в PATH и с окружением envp	process.h, stdio.h
spawnv	int spawnv(int mode, char *path, char *argv[]) Выполняет в режиме mode порожденный процесс path с аргументами argv[]	process.h, stdio.h
spawnve	int spawnve(int mode, char *path, char *argv[], char *envp[]) Выполняет в режиме mode порожденный процесс path с аргументами argv[] и с окружением envp	process.h, stdio.h
spawnvp	int spawnvp(int mode, char *path, char *argv[]) Выполняет в режиме mode порожденный процесс path с аргументами argv[] , с поиском в PATH	process.h, stdio.h
spawnvpe	int spawnvpe(int mode, char *path, char *argv[], char *envp[]) Выполняет в режиме mode порожденный процесс path с аргументами argv[] , с поиском в PATH и с окружением envp	process.h, stdio.h
system	int system(const char *command) Выполняет команду command операционной системы и возвращается в приложение	stdlib.h
wait	int wait(int *statloc) Обеспечивает ожидание завершения одного или более порожденных процессов, заносит в statloc статус завершения, возвращает ID порожденного процесса или -1	process.h

Комментарии

Функции **exec...** загружают в память и выполняют некоторую внешнюю программу **path**, называемую порожденным процессом. Вызванная программа замещает в памяти вызвавший ее процесс. Таким образом, родительский процесс завершается и начинается новый.

Различия между функциями **exec...** определяются их суффиксами, которые обозначают следующее:

- 1 В процесс передается список указателей на аргументы **arg0, arg1, ..., argn**. Обычно используется, если число аргументов заранее известно

- v** В процесс передается указатель **argv[]** на массив указателей на аргументы **arg0, arg1, ..., argn**. Обычно используется, если число передаваемых аргументов может изменяться
- p** Файл загружаемой программы ищется в каталогах, указанных в переменной окружения **PATH**. Если параметр **path** не содержит явного указания каталога, поиск ведется сначала в текущем каталоге, а затем в каталогах, указанных в **PATH**. Если функция не содержит суффикса **p**, то файл ищется только в рабочем каталоге
- e** В порождаемый процесс может быть передан аргумент **env**, указывающий на окружение порождаемого процесса. Если функция не содержит суффикса **e**, то порождаемый процесс наследует окружение родительского процесса.

Каждая из функций **exec...** должна передать в порождаемый процесс хотя бы один аргумент (**arg0**), и по соглашению этот аргумент — копия **path**. Впрочем, передача другого значения не является ошибкой. Суммарная длина всех аргументов (не учитывая нулевых символов, но учитывая пробелы) не должна превышать 128 символов.

В функциях с суффиксом **l** аргументы перечисляются непосредственно в операторе вызова функции как указатели на строки с нулевым символом в конце. Количество аргументов не ограничено. Последним аргументом передается **NULL**, что является признаком окончания списка.

В функции с суффиксом **v** в качестве параметра передается указатель на массив произвольной длины, содержащий указатели на строки, являющиеся аргументами порождаемого процесса. Последним из указателей в массиве должен быть **NULL**, показывающий, что список аргументов завершился.

В функции с суффиксом **e** передается массив указателей **env** на строки, определяющие переменные окружения порождаемого процесса. Эти строки обычно имеют вид

<имя_переменной> = <значение>

Если **env = NULL**, то для функций с суффиксом **e** так же, как и для всех остальных функций, порождаемый процесс наследует окружение родительского процесса.

Файлы, открытые на момент вызова порождаемого процесса, остаются открытыми и для этого процесса. Однако, в порожденный процесс не передается режим, в котором открыты файла (текстовый или двоичный). Если режим отличается от принятого по умолчанию, то в порожденном процессе надо произвести его установку функциями, рассмотренными в разделах 15.5.2 и 15.5.3.

Поиск файла **path**, загружаемого функциями **exec...**, осуществляется следующим образом. Если в параметре **path** явно указано расширение файла или стоит точка, ищется файл такой, который задан. Если же расширение не задано, то сначала ищется файл такой, который задан. Если он не находится, к имени добавляется расширение **.exe** и поиск повторяется. Если файл опять не находится, к имени добавляется расширение **.com** и поиск повторяется. Функции без суффикса **p** ведут поиск файла только в текущем каталоге (если только каталог не задан явно в **path**). А функции с суффиксом **p** сначала ведут поиск в текущем каталоге, а затем — в каталогах, указанных в переменной окружения **PATH**.

Все функции возвращают 0 при успешной загрузке порожденного процесса, а при ошибке возвращают -1. В этом случае глобальная переменная **errno** (см. раздел 15.1.5.1) может принимать значения **EACCES** — нарушение права доступа, **EMFILE** — слишком много открытых файлов, **ENOENT** — не найден путь или файл, **ENOEXEC** — ошибка формата, **ENOMEM** — не хватает памяти.

Приведем примеры. Оператор

```
if(exec1("F1.exe", "F1.exe", NULL))
    ShowMessage("Программа F1.exe не выполнена");
```

завершает текущий процесс и передает управление программе с выполняемым файлом **F1.exe**. Этот файл должен быть расположен в рабочем каталоге. Иначе функция **exec1** вернет -1 и будет выдано сообщение функцией **ShowMessage**. Аналогичное сообщение будет выдано если, например, для загрузки **F1.exe** не хватает оперативной памяти.

Оператор

```
exec1p("nc", "nc", NULL);
```

передает управление программе Norton Commander (файл **nc.exe**), если только путь к этой программе указан в переменной окружения **PATH**.

Оператор

```
char * prog = "command.com";
exec1p(prog, prog, NULL);
```

передает управление DOS, если только путь к файлу **command.com** указан в переменной окружения **PATH**.

Оператор

```
exec1p("Winword", "Winword", "F.doc", NULL)
```

запускает редактор Word и передает в него файл **F.doc**.

Вызов редактора Word можно оформить иначе:

```
char *arg[5] = {"Winword"}; // может принять до трех аргументов
arg[1] = "F1.doc";
arg[2] = "F2.doc";
execvp(arg[0], arg);
```

В массив **arg** при его объявлении заносится в качестве нулевого аргумента имя программы «Winword», а остальные четыре элемента массива по умолчанию получают значения **NULL**. После этого в элементы с индексами 1 и 2 заносятся имена передаваемых в Word файлов. Следующий элемент остается прежним — **NULL**. В результате функция **execvp** передаст управление программе Winword и загрузит в редактор два указанных файла.

Прежде, чем рассматривать функции **spawn...**, обладающие широкими возможностями, надо обсудить две вспомогательные функции: **cwait** и **wait**.

Функция **wait** обеспечивает ожидание завершения порожденного процесса или нескольких процессов. Окончания процессов, запущенных из этих порожденных процессов с вытеснением родителей функция не ждет.

Если параметр **statloc** функции **wait** не **NULL**, то он указывает на целое, представляющее собой статус завершения порожденного процесса. При нормальном его завершении биты этого целого означают следующее:

биты 0-7	Нули
биты 8-15	Старшие разряды кода возврата порожденного процесса. Это то значение, которое передает программа в функцию exit или в оператор return функции main . Если порожденный процесс просто покинул main без оператора return , то значение этих битов не определено

При аварийном завершении порожденного процесса биты его статуса означают:

биты 0-7	1	неисправимая ошибка
	2	генерация исключения
	3	прерывание внешним сигналом
биты 8-15	Нули	

При нормальном завершении функция **wait** возвращает идентификатор порожденного процесса. При неудаче возвращается -1, а переменная **errno** равна **EINTR** — ненормальное завершение процесса, или **ECHILD** — порожденного процесса нет.

Функция **cwait** подобна **wait**, но дает большую гибкость. Помимо параметра **statloc**, рассмотренного выше она имеет еще два параметра: **pid** и **action**. Если параметр **pid** задан равным 0, это означает, что происходит ожидание окончания любого порожденного процесса. Но в качестве значения **pid** может быть задан идентификатор конкретного порожденного процесса. Тогда происходит ожидание завершения именно указанного процесса.

Параметр **action** может принимать одно из двух значений: **WAIT_CHILD** — ожидание окончания указанного дочернего процесса, или **WAIT_GRANDCHILD** — ожидание окончания не только самого порожденного процесса, но и всех дочерних процессов, порожденных им.

Теперь можно рассмотреть функции **spawn...**. Они подобны рассмотренным функциям **exec...**, но обладают более широкими возможностями благодаря наличию параметра **mode**, задающего режим выполнения порождаемого процесса. Этот параметр может принимать следующие значения:

P_WAIT	Родительский процесс ждет завершения порожденного процесса, после чего продолжается выполнение родительского процесса
P_NOWAIT	Родительский процесс продолжает выполняться пока выполняется порожденный процесс. Поскольку функция возвращает ID порожденного процесса, можно применить функцию cwait или wait , чтобы обеспечить ожидание завершения порожденного процесса. Этот режим недоступен в 16-разрядных Windows и DOS
P_NOWAITO	Идентичен P_NOWAIT , но ID порожденного процесса не сохраняется операционной системой, так что применение функций cwait или wait невозможно
P_DETACH	Идентичен P_NOWAITO , но порожденный процесс выполняется в фоновом режиме, так что не имеет доступа к клавиатуре и дисплею
P_OVERLAY	Порождаемый процесс замещает в памяти родительский. То же, что вызов соответствующей функции exec...

• Приведем пример. Операторы

```
if(spawnlp(P_WAIT,"arj","arj","e doc.arj a1.txt", NULL))
    ShowMessage("Программа arj не выполнена");
else
{
    Memol->Clear();
    Memol->Lines->LoadFromFile("a1.txt");
    DeleteFile("a1.txt");
}
```

запускают архиватор **arj**, извлекающий из архива **doc.arj** файл **al.txt**. Приложение ждет, пока программа **arj** закончит работу, затем загружает разархивированный файл в окно редактирования **Memo1** и удаляет этот файл с диска.

В приведенном примере все аргументы, передаваемые в порождаемый процесс, объединены в одной строке. Тот же самый результат получился бы, если передать их все в отдельности:

```
if(spawnlp(P_WAIT,"arj","arj","e","doc.arj","al.txt", NULL))
...
```

Операции, подобные рассмотренным выше, невозможно было бы выполнить функциями **exec...**, поскольку они не обеспечивают возвращения в исходное приложение. Нельзя было бы выполнить эти операции и функциями **spawn...** при режиме, отличном от **P_WAIT**, поскольку в этом случае оператор загрузки файла в окно редактирования выполнялся бы раньше, чем успевал распаковываться архив. Впрочем, можно было бы использовать и режим **P_NOWAIT**, но с добавлением функций **cwait** или **wait**:

```
int ID = spawnlp(P_NOWAIT,"arj","arj","e doc.arj al.txt",
                NULL);
if(ID == -1)
    ShowMessage("Программа arj не выполнена");
else
{
    if(wait(NULL) != ID)
        ShowMessage("Ошибка разархивации");
    else
    {
        Memo1->Clear();
        Memo1->Lines->LoadFromFile("al.txt");
        DeleteFile("al.txt");
    }
}
```

В этом коде функция **spawnlp** выполняется в режиме **P_NOWAIT**, не обеспечивающем ожидание конца порожденного процесса. Но затем вызывается функция **wait**, которая обеспечивает ожидание. Если эта функция вернет значение, отличное от идентификатора порожденного процесса, значит при выполнении порожденного процесса произошло его аварийное завершение.

Надо отметить, что приведенный выше пример разархивации файла обладает двумя недостатками. Первый из них связан с тем, что выполняется программа **arj**, предназначенная для DOS. Поэтому при ее выполнении вызывается сеанс DOS, и после его окончания пользователь видит окно DOS, которое ему надо закрыть, чтобы продолжить работу. Это, конечно, очень неудобно. Устранить этот недостаток легко, например, написанием пакетного файла **arj.bat** вида:

```
@echo off
arj.exe e doc %1
exit
```

В нем помимо команда разархивации предусмотрена команда **exit** — окончание сеанса работы с окном DOS. Тогда обращение к разархивации в приложении может быть даже короче, чем раньше:

```
if(spawnlp(P_WAIT,"arj.bat","arj.bat","al.txt", NULL))
...
```

Обращение к пакетному файлу **arj.bat** позволяет порожденному процессу автоматически, без вмешательства пользователя вернуться в родительский процесс. Остается еще один недостаток рассмотренного примера — на время выполнения разархивации получают неприятные изменения экрана, связанные с выходом в DOS. Но этот недостаток может быть снят только функциями, рассмотренными в разделе 15.6.3.

Приведем еще один пример использования функции **spawnlp**. Пусть вы разработали пользовательский интерфейс, в котором хотите предоставить пользователю возможность запускать различные приложения. Ваш интерфейс достаточно большой и поэтому желательно запускать из него внешние приложения в оверлейном режиме. Эту задачу можно решить следующим образом.

Пусть имя вашего приложения **POverlay**. Создайте еще одно приложение, названное, например, **OMenage**. Это приложение будет управлять запуском требуемых программ. Оно может быть очень маленьким, не содержать ни одной формы и располагаться в оперативной памяти одновременно с запускаемыми программами. Весь текст его файла следующий:

```
#include <vcl.h>
#pragma hdrstop
#include <process.h>
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR lpCmdLine, int)
{
    spawnlp(P_WAIT, lpCmdLine, lpCmdLine, NULL);
    spawnlp(P_OVERLAY, "POverlay.exe", "POverlay.exe", NULL);
    return 0;
}
```

Первый вызов функции **spawnlp** обеспечивает запуск в режиме ожидания того приложения, имя которого передано через командную строку **lpCmdLine**. Второй вызов **spawnlp** обеспечивает оверлейный вызов вашего основного приложения **POverlay.exe**.

Предположим, что в вашем основном приложении **POverlay** имя запускаемой программы записано в окне редактирования **Edit1**. Тогда вызов этой программы может осуществляться оператором:

```
if (spawnlp(P_OVERLAY, "OMenage.exe", "OMenage.exe", Edit1->Text,
            NULL))
    ShowMessage("Программа " + Edit1->Text + " не выполнена;" +
               " нет файла OMenage.exe");
```

Этот оператор прервет выполнение приложения **POverlay** и загрузит на его место в памяти короткую (примерно 10 К) программу **OMenage.exe**, передав в нее как параметр имя запускаемого приложения. Программа **OMenage.exe** вызовет в режиме ожидания эту программу, а по окончании ее работы удалится из памяти и опять вызовет основное приложение **POverlay**. Таким образом, во время выполнения вызываемой программы в памяти будет находиться не ваше большое приложение **POverlay**, а только маленькая программа управления **OMenage.exe**.

Описанное взаимодействие программ имеет некоторый недостаток: при возврате в **POverlay** текст в окне **Edit1** будет утерян. Этот недостаток легко устранить. Измените основной файл приложения **POverlay** следующим образом:

```
#include <vcl.h>
#pragma hdrstop
USERES("POverlay.res");
USEFORM("UOverlay1.cpp", Form1);
#include "UOverlay1.h" // включение головного файла приложения

//-----
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR lpCmdLine, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TForm1), &Form1);
        Form1->Edit1->Text = lpCmdLine; // Загрузка окна Edit1
        Application->Run();
    }
}
```

```

catch (Exception &exception)
{
    Application->ShowException(&exception);
}
return 0;
}

```

По сравнению со стандартным файлом, созданным C++Builder, в него добавлено две строки (отмечены комментариями): директива, включающая головной файл модуля **UOverlay1.h**, содержащего описание вашей формы **Form1**, и оператор, загружающий в окно **Edit1** текст, переданный через командную строку. Еще одно изменение по сравнению со стандартным файлом — введение в заголовок функции **WinMain** параметра **lpCmdLine** — ссылки на командную строку. Если в файле приложения **POverlay** сделаны такие изменения, то в приложении **OMenage** второй вызов функции должен быть изменен на следующий:

```

spawnlp(P_OVERLAY, "POVERLAY.exe", "POVERLAY.exe", lpCmdLine,
        NULL);

```

Этот вызов отличается от того, что был раньше, передачей в программу той командной строки, которая была задана при вызове **OMenage**. Таким образом, в программу **POverlay** вернется имя запускавшейся программы, которое будет загружено в окно **Edit1**.

Функция **system** выполняет команду **command** и возвращает управление в вызвавшее приложение. Команда **command** может быть командой операционной системы, командой выполнения программы DOS или пакетного (batch) файла. Программа должна быть в текущем каталоге или в одном из каталогов, перечисленных в переменной окружения **PATH**. Команда выполняется командным процессором DOS, что вызывает в Windows в ряде случаев неприятное изменение экрана на время выполнения команды. Функция **system** возвращает 0 при успешном начале работы командного процессора и -1 в случае неудачи. Приведем примеры:

```

// команда DOS dir с занесением результатов
// в текстовый файл dir.txt
system("dir >> dir.txt");

// команда DOS mkdir, создающая каталог c:\\ttt
system("mkdir c:\\ttt");

// выполнение Norton Commander
system("nc");

```

15.6.3 Функции API Windows для запуска из приложения внешних программ

Методика запуска из приложения внешних программ изложена в главе 6 в разделе 6.1. Ниже даются справочные сведения по упоминаемым там процедурам и функциям API Windows.

15.6.3.1 Сообщения об ошибках при запуске внешних программ

Если описанные далее функции возвращают значение меньшее или равное 32, это указывает на ошибку. Значения ошибок означают следующее (в Windows 95, 98, 2000 и NT для некоторых из этих ошибок имеются именованные константы):

Значение	Именованная константа	Пояснение
0		Системе не хватает памяти, выполняемый файл испорчен или произошло ошибочное перераспределение памяти.

Значение	Именованная константа	Пояснение
2	ERROR_FILE_NOT_FOUND	Файл не найден.
3	ERROR_PATH_NOT_FOUND	Путь не найден.
5	SE_ERR_ACCESS-DENIED	Была попытка динамически связаться с задачей, была ошибка многопроцессорного выполнения или ошибка защиты сети.
6		Библиотека требует отдельных сегментов данных для каждой задачи.
8	SE_ERR_OOM	Недостаточно памяти для запуска приложения.
10		Ошибочная версия Windows.
11	ERROR_BAD_FORMAT	Ошибочный выполняемый файл. Или это не приложение Windows, или ошибка в .exe файле.
12		Приложение спроектировано для другой операционной системы.
13		Приложение спроектировано для MS-DOS 4.0.
14		Неизвестный тип выполняемого файла.
15		Попытка запустить приложение, работающее только на более ранних версиях Windows.
16		Попытка запустить второй экземпляр приложения, содержащего сегменты данных, не помеченные «только для чтения».
19		Попытка запустить архивированный файл. Файл должен быть разархивирован, прежде чем его можно будет загрузить.
20		Ошибочный файл одной из DLL, требуемой для приложения.
21		Приложение требует 32-битного расширения Windows.
26	SE_ERR_SHARE	Нарушение права доступа.
27	SE_ERR_ASSOCINCOMPLETE	Файл, связанный с указанной операцией не полный или ошибочный.
28	SE_ERR_DDETIMEOUT	Транзакция DDE не может быть выполнена из-за нехватки времени.
29	SE_ERR_DDEFAIL	Транзакция DDE закончилась ошибкой
30	SE_ERR_DDEBUSY	Транзакция DDE не может быть выполнена, поскольку выполняется другая транзакция DDE.

Значение	Именованная константа	Пояснение
31	SE_ERR_NOASSOC	Нет приложения, связанного с файлом указанного типа, или нет файла, связанного с указанной операцией.
32	SE_ERR_DLLNOT- FOUND	Не найдена библиотека DLL.

15.6.3.2 Функция ShellExecute

Открывает или печатает указанный файл или открывает указанную папку

Модуль *ShellAPI*

Определение

```
HINSTANCE ShellExecute(
    HWND hwnd,           // дескриптор родительского окна
    LPCTSTR lpOperation,  // строка выполняемой операции
    LPCTSTR lpFile,       // строка с именем файла или папки
    LPCTSTR lpParameters, // строка параметров выполняемого файла
    LPCTSTR lpDirectory,  // строка каталога по умолчанию
    INT nShowCmd          // режим открытия файла
);
```

Описание

Функция **ShellExecute** позволяет выполнить любое приложение Windows. Можно также открыть файл документа, что означает выполнение связанного с ним приложения и загрузку в него этого документа. Например, обычно с документами, имеющими расширение **.doc**, связан Word. В этом случае открыть файл, например, с именем «file.doc» означает запустить Word и передать ему в качестве параметра имя файла «file.doc». Кроме описанных возможностей функция **ShellExecute** позволяет распечатать указанный файл или открыть указанную папку. Последнее означает, что будет запущена программа «Проводник» с открытой указанной папкой.

Параметры функции означают следующее:

Параметр	Описание
hwnd	Родительское окно, в котором отображаются сообщения запускаемого приложения.
lpOperation	Указывает на строку с нулевым символом в конце, которая определяет выполняемую операцию. Эта строка может содержать текст «open» (открыть) или «print» (напечатать). Для Windows 95, 98 и NT определено еще одно значение: «explore» (исследовать) — открыть папку. Если параметр lpOperation равен NULL, то по умолчанию выполняется операция «open».
lpFile	Указывает на строку с нулевым символом в конце, которая определяет имя открываемого файла.
lpParameters	Указывает на строку с нулевым символом в конце, которая определяет передаваемые в приложение параметры, если File-Name определяет выполняемый файл. Если lpFile указывает на строку, определяющую открываемый документ, то этот параметр задается равным NULL.
lpDirectory	Указывает на строку с нулевым символом в конце, которая определяет каталог по умолчанию.

Параметр	Описание	
nShowCmd	Определяет, режим открытия указанного файла. Этот параметр может иметь значения:	
	SW_HIDE	Окно делается невидимым и фокус передается другому окну
	SW_MINIMIZE	Свертывает (минимизирует) указанное окно и активизирует следующее в Z-последовательности окно верхнего уровня в списке системы
	SW_MAXIMIZE	Развертывает (максимизирует) указанное окно
	SW_RESTORE	Активизирует и отображает окно. Если это окно свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции (почти то же самое, что SW_SHOWNORMAL)
	SW_SHOW	Активизирует и отображает окно в его текущей позиции и с текущими размерами
	SW_SHOW-DEFAULT	Только для Windows 95, 98, 2000 и NT. Устанавливает состояние в соответствии с флагом SW_ в структуре STARTUPINFO , передаваемой в функцию CreateProcess программой, запускающей приложение. Приложение должно вызывать ShowWindow с этим флагом, чтобы задать начальное состояние своего главного окна
	SW_SHOW-MAXIMIZED	Активизирует и отображает окно в развернутом виде (максимизированном)
	SW_SHOW-MINIMIZED	Активизирует и отображает окно в свернутом виде (в виде пиктограммы)
	SW_SHOW-MINNOACTIVE	Отображает окно в свернутом виде (в виде пиктограммы). Активным остается то окно, которое было активным до этого
	SW_SHOWNA	Отображает окно в его текущей позиции и с текущими размерами. Активным остается то окно, которое было активным до этого
	SW_SHOW-NOACTIVATE	Отображает окно в его последней позиции и с последними размерами. Активным остается то окно, которое было активным до этого
	SW_SHOW-NORMAL	Активизирует и отображает окно. Если это окно свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции (почти то же самое, что SW_RESTORE)

Функция возвращает дескриптор открытого приложения или дескриптор сервера DDE приложения. Если возвращаемое значение меньше или равно 32, это указывает на ошибку. Значения ошибок приведены в таблице раздела 15.6.3.1.

Примеры

1. Пусть вы хотите открыть файл документа с именем «file.doc», т.е. запустить Word (обычно именно он связан с файлами .doc), загрузив в него указанный файл. Тогда вы можете написать оператор:

```
ShellExecute(Handle, NULL, "file.doc", NULL, NULL, SW_RESTORE);
```

2. Печать документа осуществляется аналогично рассмотренному выше, только надо задать соответствующее значение параметра **lpOperation**:

```
ShellExecute(Handle, "print", "file.doc", NULL, NULL, SW_RESTORE);
```

Выполнение этого оператора будет протекать следующим образом. Запустится Word, связанный с файлами .doc, в него загрузится файл «file.doc», затем из Word запустится печать с атрибутами по умолчанию, после чего файл «file.doc» выгрузится из Word.

3. Ниже дан пример открытия приложения «Калькулятор»:

```
ShellExecute(Handle, "open", "Calc", NULL, NULL, SW_RESTORE);
```

4. Следующий пример открывает папку c:\Program Files\Borland:

```
ShellExecute(Handle, "open", "c:\\Program Files\\Borland",  
NULL, NULL, SW_RESTORE);
```

а оператор

```
ShellExecute(Handle, "explore",  
"c:\\Program Files\\Borland",  
NULL, NULL, SW_RESTORE);
```

открывает программу «Проводник» с открытой папкой c:\Program Files\Borland.

15.6.3.3 Функция FindExecutable

Возвращает имя и путь приложения, связанного с указанным файлом

Модуль *ShellAPI*

Определение

```
HINSTANCE FindExecutable(  
    LPCTSTR lpFile,           // строка с именем файла документа  
    LPCTSTR lpDirectory,     // строка каталога по умолчанию  
    LPTSTR lpResult           // строка с именем выполняемого файла  
);
```

Описание

Функция **FindExecutable** позволяет получить имя выполняемого файла .exe, связанного с файлом, указанным параметром **lpFile**. Параметр **lpDirectory** определяет каталог по умолчанию. Оба параметра являются указателями на строки с нулевым символом в конце. Параметр **lpResult** является указателем на буфер в виде строки с нулевым символом в конце, в который функция заносит имя и путь приложения, связанного с файлом **lpFile**.

При успешном завершении функция **FindExecutable** возвращает значение, большее 32. Если возвращено меньшее значение, это свидетельствует об ошибке. Значения ошибок приведены в таблице раздела 15.6.3.1.

Пример

Операторы

```
char APchar[254];  
FindExecutable("Doc.doc", NULL, APchar);
```

```
Label1->Caption = APchar;
```

приведут к тому, что в метку **Label1** будет занесено имя приложения, связанного с файлом типа **.doc**, например:

```
C:\MSOFFICE\WINWORD\WINWORD.EXE
```

15.6.3.4 Функция WinExec

Запускает указанное приложение

Определение

```
UINT WinExec(  
    LPCSTR lpCmdLine, // адрес командной строки  
    UINT uCmdShow      // режим открытия приложения  
);
```

Описание

Функция **WinExec** позволяет выполнить указанное приложение. Параметр **lpCmdLine** является указателем на строку с нулевым символом в конце, содержащую имя выполняемого файла и, если необходимо, параметры командной строки.

Если имя указано без пути, то Windows ищет выполняемый файл в следующей последовательности:

1. Каталог, из которого загружено приложение.
2. Текущий каталог.
3. Системный каталог Windows, возвращаемый функцией **GetSystemDirectory**.
4. Каталог Windows, возвращаемый функцией **GetWindowsDirectory**.
5. Список каталогов из переменной окружения **PATH**.

Параметр **uCmdShow** определяет форму представления окна запускаемого приложения Windows. Возможные значения этого параметра см. в разделе 15.6.3.2. Для приложений не Windows, для файлов PIF и т.д. состояние окна определяет само приложение.

При успешном выполнении запуска приложения функция **WinExec** возвращает значение, большее 31. Если возвращено меньшее значение, это свидетельствует об ошибке. Значения ошибок приведены в таблице раздела 15.6.3.1.

Достоинством функции **WinExec** является ее совместимость с ранними версиями Windows. Собственно для этого она и сохраняется в WIN32, хотя для Win32 рекомендуется пользоваться функцией **CreateProcess** (об этой функции см. встроенную в C++Builder справку).

При работе с Win32 функция **WinExec** завершает работу, если вызванное приложение вызывает функцию **GetMessage** или заканчивается выделенный лимит времени. Таким образом, ожидание можно прервать, предусмотрев в процессе, запущенном с помощью **WinExec**, в нужный момент вызов функции **GetMessage**.

Примеры

Оператор

```
WinExec("file.exe", SW_RESTORE);
```

запускает программу **file.exe**. Оператор

```
WinExec("nc", SW_RESTORE);
```

запускает Norton Commander. Оператор

```
WinExec("COMMAND.COM", SW_RESTORE);
```

приводит к запуску MS-DOS.

15.7 Функции различного назначения

15.7.1 Функции динамического распределения памяти

Функция	Синтаксис / Описание	Файл
_msize	size_t _msize(void *block) Возвращает размер блока с указателем block , выделенного ранее функциями malloc , calloc , realloc ; только для 32-разрядных приложений	malloc.h
_new_handler	typedef void (*pvf)(); pvf _new_handler Указатель на функцию, вызываемую при невозможности выделить память операцией new	new.h
alloca	void *alloca(size_t size) Выделяет пространство размером size в стеке; возвращает указатель на него или NULL	malloc.h
AllocMem	void * AllocMem(Cardinal Size) Динамически выделяет область памяти размером Size байтов и возвращает указатель (void *) на выделенную область; эта область в дальнейшем может быть освобождена процедурой FreeMem	SysUtils.hpp
calloc	void *calloc(size_t nitems, size_t size) Выделяет память под nitems элементов размером size каждый; возвращает указатель на выделенный блок памяти или NULL	stdlib.h
free	void free(void *block) Освобождает блок памяти block , выделенный ранее функциями calloc , malloc , realloc	stdlib.h
GetMemoryManager	void GetMemoryManager(TMemoryManager &MemMgr) Возвращает указатель MemMgr на функции пользователя, выделяющие и освобождающие память	System.hpp
malloc	void *malloc(size_t size) Выделяет блок памяти размером size ; возвращает указатель на этот блок или NULL	stdlib.h или alloc.h
realloc	void *realloc(void *block, size_t size) Изменяет размер блока block , выделенного ранее функциями malloc , calloc , realloc , на size ; возвращает указатель на выделенный блок памяти или NULL	stdlib.h
Set_new_handler	typedef void (new * new_handler)(); new_handler set_new_handler(new_handler my_handler) Устанавливает функцию my_handler , которая будет вызываться при невозможности выделить память операцией new	new.h

Функция	Синтаксис / Описание	Файл
SetMemoryManager	void SetMemoryManager(const TMemoryManager &MemMgr) Устанавливает параметром MemMgr функции пользователя, выделяющие и освобождающие память	System.hpp
SysFreeMem	extern PACKAGE int SysFreeMem(void * P) Освобождает память, выделенную под блок с указателем P заказным диспетчером памяти	System.hpp или ShareMem.hpp
SysGetMem	extern PACKAGE void * SysGetMem(int Size) Выделяет блок памяти размером Size , если введен заказной диспетчер памяти; возвращает указатель на блок или NULL	System.hpp или ShareMem.hpp
SysReallocMem	extern PACKAGE void * SysReallocMem(void * P, int Size) Изменяет размер блока с указателем P до размера Size , если введен заказной диспетчер памяти; возвращает указатель на блок или NULL	System.hpp или ShareMem.hpp
THeapStatus	System::THeapStatus GetHeapStatus(void) Заносит информацию о состоянии heap в структуру типа THeapStatus	System.hpp или ShareMem.hpp

Комментарии

Для функций, в которых в приведенной таблице указано два заголовочных файла — **System.hpp** или **ShareMem.hpp**, файл **System.hpp** надо подключать, если динамически распределяется глобальная область памяти, а файл **ShareMem.hpp** надо подключать, если динамически распределяется область памяти, которую могут совместно использовать различные процессы.

Для динамического распределения выделяется специальная область памяти — **heap**. Динамическое распределение памяти в этой области может производиться несколькими способами: с помощью библиотечных функций **malloc**, **calloc**, **realloc**, **free** или с помощью операций **new** и **delete** (см. в главе 12 в разделе 12.9).

Функция **malloc** выделяет в **heap** блок размером в **size** байтов. В случае успешного выделения памяти функция возвращает указатель на выделенный блок. Если не хватило места для блока требуемого размера или если **size = 0**, возвращается **NULL**.

Другая функция — **calloc** выделяет память под **nitems** объектов, размер каждого из которых равен **size**. Таким образом общий объем выделяемой памяти составляет **nitems * size**. Выделенная память инициализируется нулями. В случае успешного выделения памяти функция возвращает указатель на выделенный блок. Если не хватило места для блока требуемого размера или если **size = 0** или **nitems = 0**, возвращается **NULL**.

Еще одна функция — **realloc** позволяет изменить размер ранее выделенного блока памяти. Она изменяет размер блока в **heap**, на который указывает **block**, до размера **size**. При этом предполагается, что **block** указывает блок памяти, выделенной ранее функциями **malloc**, **calloc** или **realloc**. Если же аргумент **block** задан равным **NULL**, то функция **realloc** работает так же, как описанная выше функция **malloc**.

Если размер **size** задан равным нулю, то выделенный ранее блок, на который указывает **block**, освобождается, а функция возвращает **NULL**. Таким образом,

функция с **size** равным 0 может использоваться не для выделения памяти, а для освобождения памяти, выделенной ранее.

Если блок нового размера не может быть выделен, то функция **realloc** возвращает **NULL**. Если же память выделилась успешно, то возвращается адрес выделенного блока. При этом он может отличаться от начального значения **block**, поскольку функция при необходимости осуществляет копирование содержимого блока в новое место.

Функция **free** освобождает блок памяти, выделенный ранее функциями **malloc**, **calloc** или **realloc**, на который указывает **block**.

Рассмотрим примеры использования описанных функций. Следующий код динамически выделяет функцией **malloc** память под строку, а затем, после выполнения с ней каких-то операций, освобождает выделенную память.

```
#include <stdio.h>
#include <alloc.h>
char *str;

// str - указатель на строку, под которую выделена память
str = (char *) malloc(100);

...
// освобождение памяти
free(str);
```

В этом примере можно было бы использовать для выделения памяти функцию **calloc**:

```
str = (char *) calloc(100, sizeof(char));
```

Размер выделенной функциями **malloc** или **calloc** памяти можно было бы изменить, например, следующим оператором:

```
str = (char *) realloc(str, 20);
```

Впрочем, к тому же результату привел бы и более простой оператор:

```
realloc(str, 20);
```

Необходимо помнить, что рассмотренные функции возвращают **NULL** (0), если память не удалось выделить. Поэтому прежде, чем использовать возвращенные ими указатели, надо обязательно проверять, не равны ли они **NULL**. Иначе возможно очень тяжелые ошибки при работе программы.

_new_handler является указателем на функцию, вызываемую при невозможности выделить память операцией **new**. По умолчанию эта функция просто завершает приложение. Но имеется возможность заменить функцию по умолчанию своей функцией. Если происходит возврат из этой функции, то делается повторная попытка динамически выделить память.

Установка новой функции, на которую указывает **_new_handler**, производится функцией **set_new_handler**. Вариант этой функции **set_new_handler(0)** устанавливает функцию по умолчанию. Собственная функция не получает никаких параметров и не возвращает никакого значения. Она может выполнять одно из следующих действий:

- вернуться после освобождения памяти (тогда будет сделана повторная попытка выделить память)
- сгенерировать исключение типа **bad_alloc** или одного из производных от **bad_alloc** классов
- завершить приложение функциями **abort** или **exit**

Функция **set_new_handler** возвращает указатель на прежнюю функцию. Это можно использовать для последующего ее восстановления. Таким образом, вызов функции установки может иметь вид:

```
pvf set_new_handler(pvf p);
```

Функция **THeapStatus** заносит информацию о состоянии памяти в структуру типа **THeapStatus**, содержащую поля:

TotalAddrSpace	Общее текущее адресное пространство в байтах, доступное программе. Увеличивается по мере увеличения динамически распределяемой памяти
TotalUncommitted	Общее число байтов в TotalAddrSpace , которое не выделено для своппируемого файла
TotalCommitted	Общее число байтов в TotalAddrSpace , которое выделено для своппируемого файла. Справедливо соотношение TotalUncommitted + TotalCommitted = TotalAddrSpace
TotalAllocated	Объем в байтах динамически выделенной в программе области памяти
TotalFree	Полное число байтов, доступное для программы. Если это число превышает и доступно достаточно виртуальной памяти, то OS увеличивает доступное адресное пространство. Соответственно увеличивается и TotalAddrSpace
FreeSmall	Число байтов небольших блоков памяти, которые могут быть еще выделены вашей программе
FreeBig	Число байтов больших блоков памяти, которые могут быть еще выделены вашей программе. Большие свободные блоки могут создаваться объединением смежных малых свободных блоков или динамическим выделением большого блока
Unused	Общее число байтов, которые не могут использоваться программой. Справедливо соотношение: Unused + FreeBig + FreeSmall = TotalFree
Overhead	Число байтов, требуемое диспетчером динамически распределяемой памяти для управления всеми блоками
HeapErrorCode	Индикатор внутреннего состояния динамически распределяемой памяти

Поля **TotalAddrSpace**, **TotalUncommitted** и **TotalCommitted** относятся к памяти, выделенной для программы системой. А поля **TotalAllocated** и **TotalFree** относятся к области динамически распределяемой памяти (в дальнейшем для краткости будем называть ее *heap*). Так что для проверки возможностей динамического выделения памяти надо ориентироваться на **TotalAllocated** и **TotalFree**.

Функция **SetMemoryManager** позволяет пользователю заменить функции, выделяющие и освобождающие память, своими собственными. Эти функции пользователя задаются полями параметра **MemMgr** типа **TMemoryManage**. Структура типа **TMemoryManage** имеет поля:

GetMem	Указывает на функцию, выделяющую в памяти блок с заданным числом байтов Size и возвращающую указатель на выделенный блок (аналог функции malloc). Параметр Size функции GetMem не должен быть равен нулю. Если GetMem не может выделить блок заданного размера, она должна возвращать NULL
---------------	---

FreeMem	Указывает на функцию, освобождающую блок памяти, на который указывает ее параметр (аналог функции free). Параметр функции FreeMem не должен быть равен NULL . Если FreeMem успешно освободила память, она должна возвращать 0. В противном случае должно возвращаться ненулевое значение
ReallocMem	Указывает на функцию, которая изменяет размер блока, на который указывает ее параметр, до заданной новой величины Size (аналог функции realloc). Указатель, передаваемый в функцию ReallocMem , не должен быть равен NULL , а параметр Size не должен быть равен 0. Функция ReallocMem должна изменить размер блока, при необходимости переместив его на новое место, если нельзя обеспечить требуемый размер на прежнем месте. Информация, хранившаяся в прежнем блоке, должна быть сохранена, но вновь выделяемое пространство может не инициализироваться. Функция должна возвращать указатель на блок или NULL , если изменить размер блока невозможно

Функции пользователя, которые устанавливаются функцией **SetMemoryManager**, могут оперировать с объектами, их конструкторами и деструкторами, строками и т.п. Функция **GetMemoryManager** возвращает структуру типа **TMemoryManager**, содержащую указатели на установленные функции. Через поля этой структуры можно обращаться к установленным функциям.

Приведем пример. Пусть вы хотите вести учет числа обращений к функциям динамического распределения памяти и учет объемов выделяемой и освобождаемой памяти. Это можно сделать следующим кодом:

```
#include <malloc.h>

TMemoryManager* mmNew;
TMemoryManager* mmOld;
long alloc, dealloc, Nalloc, Ndealloc;

...
void * __fastcall NewGetMem(int Size)
{
    alloc += Size;
    Nalloc++;
    return mmOld->GetMem(Size);
}

int __fastcall NewFreeMem(void *p)
{
    dealloc = _msize(p);
    Ndealloc ++;
    return mmOld->FreeMem(p);
}

void * __fastcall NewReallocMem(void *p, int Size)
{
    alloc += Size;
    dealloc = _msize(p);
    Nalloc++;
    Ndealloc ++;
    return mmOld->ReallocMem(p, Size);
}

__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{

```

```

mmNew = new TMemoryManager();
mmOld = new TMemoryManager();
mmNew->GetMem = NewGetMem;
mmNew->FreeMem = NewFreeMem;
mmNew->ReallocMem = NewReallocMem;
GetMemoryManager(*mmOld);
SetMemoryManager(*mmNew);
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Label1->Caption = "Nalloc = "+IntToStr(Nalloc)+
                    "Ndealloc = "+IntToStr(Ndealloc)+
                    " выделено " +IntToStr(alloc)
                    " освобождено " +IntToStr(dealloc);
}

```

В этом примере при щелчке на кнопке **Button1** в метке **Label1** отображается сообщение о динамическом распределении памяти.

15.7.2 Функции вызова диалоговых окон с сообщениями

15.7.2.1 Процедуры ShowMessage и ShowMessageFmt

В приложениях часто приходится отображать различные простые диалоговые окна, чтобы дать пользователю какие-то указания или задать несложный вопрос, на который возможен один из стандартных ответов: да, нет, отменить, прервать. В законченном приложении желательно эти окна проектировать самому, обеспечивая единство стиля всех окон приложения, русские надписи на кнопках и т.п. Но при разработке прототипа будущего проекта и в процессе отладки удобно пользоваться готовыми диалоговыми окнами и вызывающими их процедурами.

Простейшей из таких процедур является **ShowMessage**, отображающая окно сообщения с кнопкой ОК. Ее объявление:

```
void ShowMessage(const System::AnsiString Msg)
```

Текст сообщения задается параметром **Msg**. Заголовок окна совпадает с именем выполняемого файла приложения.

Имеется также похожая процедура **ShowMessageFmt**, позволяющая выводить в аналогичное окно форматированное сообщение. Объявление этой процедуры имеет вид:

```
void ShowMessageFmt(const System::AnsiString Msg,
                    const System::TVarRec *Params, const int Params_Size)
```

Параметр **Msg** в этой процедуре задает строку описания формата (см. раздел 15.1.4.3), а параметры **Params** и **Params_Size** задают массив параметров, формируемых строкой **Msg**, и размер этого массива. Для передачи массива в функцию удобно использовать макрос **OPENARRAY** (см. раздел 15.7.4). Тогда вызов функции **ShowMessageFmt** имеет вид:

```
ShowMessageFmt(Msg, OPENARRAY(TVarRec, (arg1, arg2, ...)));
```

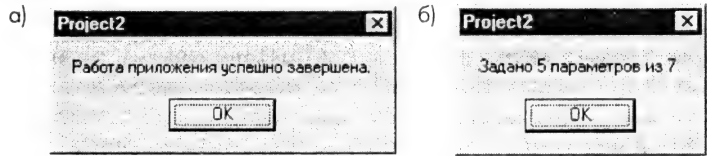
Приведем примеры использования этих процедур.

```
ShowMessage("Работа приложения успешно завершена.");
```

```
ShowMessageFmt("Задано %d параметров из %d ",
                OPENARRAY(TVarRec, (N1, N2)));
```

На рис. 15.2 а показано окно, создаваемое первым из этих операторов, а на рис. 7.1 б — вторым при **N1 = 5**, **N2 = 7**. Еще один пример окна с несколькими строками табулированного текста вы можете увидеть на рис. 15.1 в разделе 15.1.3.

Рис. 15.2.
Сообщения, выдаваемые функциями ShowMessage (a) и ShowMessageFmt (б)



15.7.2.2 Функции MessageDlg, MessageDlgPos и CreateMessageDialog

Рассмотренные в разделе 15.7.2.1 процедуры отображают окна, в которых пользователь, прочтя текст, должен просто нажать ОК. Следующие функции отображают окна, в которых пользователю задается какой-то вопрос и анализируется полученный ответ. Основная из этих функций — **MessageDlg**. Она объявлена следующим образом:

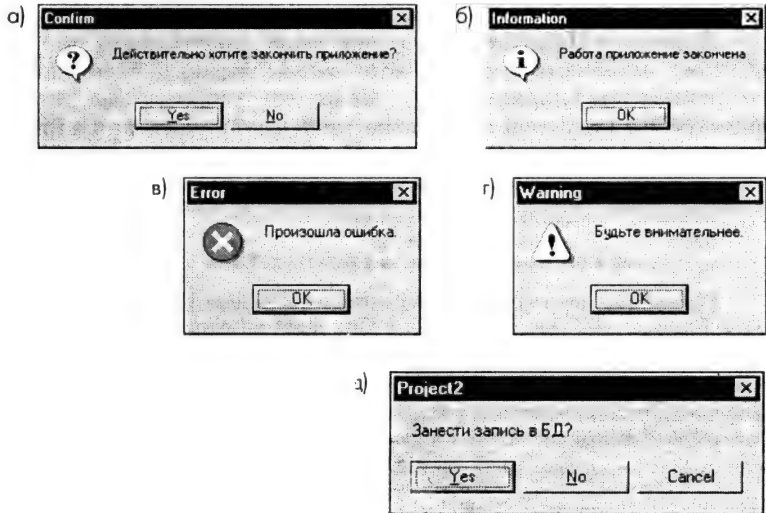
```
int MessageDlg(const System::AnsiString Msg,
               TMsgDlgType DlgType,
               TMsgDlgButtons Buttons, int HelpCtx)
```

Вызов **MessageDlg** отображает диалоговое окно и ожидает ответа пользователя. Сообщение в окне задается параметром функции **Msg**.

Вид отображаемого окна задается параметром **DlgType**. Возможные значения этого параметра:

Значение	Описание
mtConfirmation	Окно подтверждения, содержащее зеленый вопросительный знак (см. рис. 15.3 а)
mtInformation	Информационное окно, содержащее голубой символ «i» (см. рис. 15.3 б)
mtError	Окно ошибок, содержащее красный стоп-сигнал (см. рис. 15.3 в)
mtWarning	Окно замечаний, содержащее желтый восклицательный знак (см. рис. 15.3 г)
mtCustom	Заказное окно без рисунка. Заголовок соответствует имени выполняемого файла приложения (см. рис. 15.3 д)

Рис. 15.3.
Примеры диалоговых окон, выводимых функциями MessageDlg и MessageDlgPos



Параметр **AButtons** определяет, какие кнопки будут присутствовать в окне. Тип **TMsgDlgBtns** параметра **AButtons** является множеством, которое включает различные кнопки. Возможные значения видов кнопок:

Значение	Описание
mbYes	Кнопка с надписью «Yes»
mbNo	Кнопка с надписью «No»
mbOK	Кнопка с надписью «OK»
mbCancel	Кнопка с надписью «Cancel»
mbHelp	Кнопка с надписью «Help»
mbAbort	Кнопка с надписью «Abort»
mbRetry	Кнопка с надписью «Retry»
mbIgnore	Кнопка с надписью «Ignore»
mbAll	Кнопка с надписью «All»

Необходимые кнопки заносятся в **Buttons** операцией «<<», поскольку параметр **Buttons** является множеством. Если не занести в этот параметр ничего, в окне не будет ни одной кнопки и пользователю придется закрывать окно системными кнопками Windows.

Кроме множества значений, соответствующих отдельным кнопкам, определены три константы, соответствующие множествам часто используемых сочетаний кнопок:

Множество	Описание
mbYesNoCancel	Включает в окно кнопки Yes, No и Cancel
mbOkCancel	Включает в окно кнопки OK и Cancel
mbAbortRetryIgnore	Включает в окно кнопки Abort, Retry и Ignore

Эти предопределенные множества имеют тип **TMsgDlgButtons** и могут непосредственно включаться в вызов функции вместо параметра **Buttons**.

Параметр **HelpCtx** определяет экран контекстной справки, соответствующий данному диалоговому окну. Этот экран справки будет появляться при нажатии пользователем клавиши F1. Если вы справку не планируете, при вызове **MessageDlg** надо задать нулевое значение параметра **HelpCtx**.

Функция **MessageDlg** возвращает значение, соответствующее выбранной пользователем кнопке. Возможные возвращаемые значения:

mrNone

mrAbort

mrYes

mrOk

mrRetry

mrNo

mrCancel

mrIgnore

mrAll

Приведем примеры использования функции **MessageDlg**. Первый пример иллюстрирует диалог при окончании работы приложения:

```
if(MessageDlg("Действительно хотите закончить приложение?",
    mtConfirmation, TMsgDlgButtons() << mbYes<< mbNo, 0) == mrYes)
{
    MessageDlg("Работа приложение закончена", mtInformation,
        TMsgDlgButtons() << mbOK, 0);
    Close();
}
```

Первый вызов **MessageDlg** приводит к отображению окна типа **mtConfirmation** с вопросом о завершении приложения (см. рис. 15.3 а). Если пользователь нажимает кнопку Yes, то выводится второе окно типа **mtInformation** с сообщением о завершении (см. рис. 15.3 б).

Следующий пример иллюстрирует диалог при генерации исключения (см. рис. 15.3 в и 15.3 г):

```
catch ( ... )
{
    MessageDlg("Произошла ошибка.", mtError, TMsgDlgButtons() << mbOK, 0);
    MessageDlg("Будьте внимательнее.", mtWarning,
               TMsgDlgButtons() << mbOK, 0);
}
```

Следующий пример иллюстрирует работу с базой данных, когда после редактирования пользователем записи ему предлагается вопрос о сохранении ее в базе данных (см. рис. 15.3 д). Если пользователь выбирает кнопку Yes, запись сохраняется методом **Post**; если пользователь выбирает кнопку No, результаты редактирования уничтожаются методом **Cancel**; если же пользователь выбирает кнопку **Cancel**, форма закрывается.

```
switch (MessageDlg("GBRDZbP HBVPZh > ;??", mtCustom, mbYesNoCancel, 0))
{
    case mrYes: Table1->Post();
                break;
    case mrNo : Table1->Cancel();
                break;
    case mrCancel : Close();
}
```

Имеется также функция **MessageDlgPos**, во всем подобная **MessageDlg**, но отображающая диалоговое окно сообщений в заданном месте экрана. Объявление этой функции:

```
int MessageDlgPos(const System::AnsiString Msg,
                  TMsgDlgType DlgType, TMsgDlgButtons Buttons,
                  int HelpCtx, int X, int Y)
```

Вызов **MessageDlgPos** отображает диалоговое окно в заданном месте экрана и ожидает ответа пользователя. Координаты определяются параметрами **X** и **Y**. Основные параметры тождественны функции **MessageDlg**. Например, оператор

```
MessageDlgPos("Будьте внимательнее.", mtWarning,
              TMsgDlgButtons() << mbOK, 0, 250, 0);
```

вызовет появление окна сообщения вверху экрана (параметр **Y=0**) примерно в центре. А оператор

```
MessageDlgPos("Ошибка в этом окне!", mtError,
              TMsgDlgButtons() << mbOK, 0,
              BoundsRect.Left, BoundsRect.Bottom);
```

отобразит диалоговое окно вблизи нижнего левого угла формы, в которой записан данный оператор.

Функция **CreateMessageDialog**, определенная следующим образом:

```
extern PACKAGE Forms::TForm* CreateMessageDialog(
    const System::AnsiString Msg, TMsgDlgType DlgType,
    TMsgDlgButtons Buttons)
```

позволяет создать диалоговое окно сообщения в виде объекта формы. Функция только создает окно, но не отображает его. Отображение осуществляется обычными для форм методами **Show** или **ShowModal**. При использовании метода **ShowModal** можно анализировать ответ пользователя так же, как это делается для любых модальных форм.

Использованные для задания типа диалога **DlgType** и кнопок окна **Buttons** типы данных **TMsgDlgType** и **TMsgDlgButtons** были описаны выше.

Функцию **CreateMessageDialog** имеет смысл применять для создания диалогового окна, которое будет использоваться в приложении многократно. При этом преимуществом этого окна по сравнению с теми, которые создавались ранее рассмотренными функциями, заключается в том, что вы можете задать русскую надпись в заголовке окна, как делаете это для любой формы. В то же время существенным недостатком применения функции **CreateMessageDialog** является то, что объект диалогового окна хранится в памяти все время, пока он не будет уничтожен явно методом **Free**. Это приводит к непроизводительным затратам памяти.

Приведем пример использования **CreateMessageDialog**. Операторы

```
TForm *FMess;
...
FMess = CreateMessageDialog("Будьте внимательнее.", mtWarning,
                           TMsgDlgButtons() << mbOK);
FMess->Caption = "Предупреждение";
```

создают объект **FMess** диалогового окна, задают текст его сообщения и заголовок «Предупреждение». Вид этого окна (рис. 15.4) идентичен приведенному ранее на рис. 15.3 г, за исключением заголовка «Предупреждение» вместо непонятного не слишком опытному пользователю заголовка «Warning». Оператор

```
FMess->ShowModal();
```

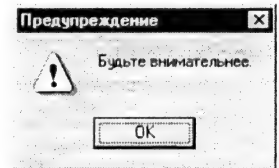
отображает окно как модальную форму. Оператор

```
FMess->Free();
```

уничтожает объект, после чего окно уже не сможет отображаться.

Рис. 15.4.

Окно, созданное функцией **CreateMessageDialog**



15.7.2.3 Функция **TApplication->MessageBox**

Основным недостатком рассмотренных в разделах 15.7.11.1 и 15.7.11.2 функций и процедур являлось отсутствие русификации диалоговых окон (английские надписи на кнопках) и невозможность указать текст заголовка окна (кроме функции **CreateMessageDialog**). Эти недостатки позволяет устранить метод **MessageBox** переменной **Application** типа **TApplication**, доступной в любом приложении **C++Builder**. Поскольку это метод компонента, он выходит за рамки тематики этой главы. Но поскольку тематически этот метод очень близок рассмотренным выше функциям, обсудим его в данном разделе.

Метод объявлен следующим образом:

```
int __fastcall MessageBox(const char * Text,
                          const char * Caption, int Flags);
```

Он отображает диалоговое окно с заданными кнопками, сообщением и заголовком и позволяет проанализировать ответ пользователя. Во многих отношениях это окно подобно окнам, создаваемым функциями **MessageDlg** и **CreateMessageDialog**. Но имеются и существенные отличия, связанные с возможностью русификации окна. Заголовок окна может быть написан по-русски, что отличает эту функцию от функции **MessageDlg** (впрочем, в окне, созданном **CreateMessageDialog**, это тоже можно сделать). Другим приятным отличием являются русские надписи на кнопках (в русифицированных версиях Windows).

Функция **MessageBox** инкапсулирует функцию **MessageBox** API Windows. Параметр **Text** представляет собой текст сообщения, которое может превышать 255 символов. Для длинных сообщений осуществляется автоматический перенос текста. Параметр **Caption** представляет собой текст заголовка окна. Он тоже может превышать 255 символов, но не переносится. Так что длинный заголовок приводит к появлению длинного и не очень красивого диалогового окна.

Параметр **Flags** представляет собой множество флагов, определяющих вид и поведение диалогового окна. Этот параметр может комбинироваться операцией сложения по одному флагу из следующих групп.

Флаги кнопок, отображаемых в диалоговом окне

Флаг	Значение (в скобках даны надписи в русифицированных версиях Windows)
MB_ABORTRETRYIGNORE	Кнопки Abort (Стоп), Retry (Повтор) и Ignore (Пропустить).
MB_OK	Кнопка ОК. Этот флаг принят по умолчанию.
MB_OKCANCEL	Кнопки ОК и Cancel (Отмена).
MB_RETRYCANCEL	Кнопки Retry (Повтор) и Cancel (Отмена).
MB_YESNO	Кнопки Yes (Да) и No (Нет).
MB_YESNOCANCEL	Кнопки Yes (Да), No (Нет) и Cancel (Отмена).

Флаги пиктограмм в диалоговом окне

Флаг	Пиктограмма
MB_ICONEXCLAMATION, MB_ICONWARNING	Восклицательный знак (замечание, предупреждение).
MB_ICONINFORMATION, MB_ICONASTERISK	Буква i в круге (подтверждение).
MB_ICONQUESTION	Знак вопроса (ожидание ответа).
MB_ICONSTOP, MB_ICONERROR, MB_ICONHAND	Знак креста на красном круге (запрет, ошибка).

Флаги, указывающие кнопку по умолчанию (которая в первый момент находится в фокусе)

Флаг	Кнопка
MB_DEFBUTTON1	Первая кнопка. Это принято по умолчанию.
MB_DEFBUTTON2	Вторая кнопка.
MB_DEFBUTTON3	Третья кнопка.
MB_DEFBUTTON4	Четвертая кнопка.

Флаги модальности

Флаг	Пояснение
MB_APPLMODAL	Пользователь должен ответить на запрос, прежде чем сможет продолжить работу с приложением. Но он может перейти в окна другого приложения. Он может также работать со всплывающими окнами данного приложения. Этот флаг принят по умолчанию.
MB_SYSTEMMODAL	То же самое, что MB_APPLMODAL , но окно диалога отображается в стиле WS_EX_TOPMOST , то есть всегда остается поверх других окон, даже если пользователь перешел к другим приложениям. Используется для предупреждения о серьезных ошибках, требующих немедленного вмешательства.

Некоторые дополнительные флаги (могут задаваться оба флага)

Флаг	Пояснение
MB_HELP	Добавляет в окно кнопку Help (Справка), щелчок на которой или нажатие клавиши F1 генерирует событие Help.
MB_TOPMOST	Помещает окно всегда сверху (в стиле WS_EX_TOPMOST).

Возможны еще некоторые флаги, определяющие характер поведения окна при работе в сети нескольких пользователей, позволяющие отображать тексты справа налево (для восточных языков) и т.п.

Функция возвращает нуль, если не хватает памяти для создания диалогового окна. Если же функция выполнена успешно, то возвращаемая величина свидетельствует о следующем:

Значение	Численное значение	Пояснение
IDABORT	3	Выбрана кнопка Abort (Стоп).
IDCANCEL	2	Выбрана кнопка Cancel (Отмена) или нажата клавиша Esc.
IDIGNORE	5	Выбрана кнопка Ignore (Пропустить).
IDNO	7	Выбрана кнопка No (Нет).
IDOK	1	Выбрана кнопка ОК.
IDRETRY	4	Выбрана кнопка Retry (Повтор).
IDYES	6	Выбрана кнопка Yes (Да).

Рассмотрим пример. Ниже приведен текст, предусматривающий проверку правильности ввода данных перед пересылкой записи в базу данных.

```

if (проверка введенных данных)
{
    if (Application->MessageBox(
        "Хотите занести текущую запись в базу данных?",
        "Подтвердите занесение в базу данных",
        MB_YESNOCANCEL + MB_ICONQUESTION) != IDYES)

```

```

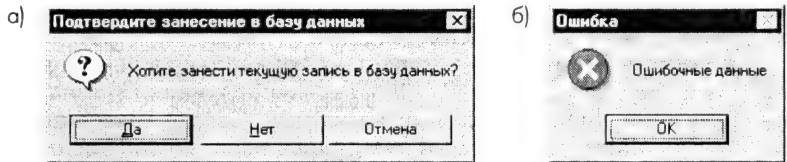
{
    DataSet->Cancel();
    Abort();
}
}
else
{
    Application->MessageBox("Ошибочные данные", "Ошибка",
                           MB_ICONSTOP);
    Abort();
}
}

```

Отображаемые этим кодом окна приведены на рис. 15.5. Безусловно они более удачны, чем приведенные в предыдущих разделах, за счет русификации.

Рис. 15.5.

Диалоговые окна, отображаемые функцией Application->MessageBox



15.7.2.4 Функции InputBox и InputQuery

Функции, описанные в этом разделе, предлагают пользователю диалоговое окно, в котором он должен ввести в окошко редактирования некоторый текст. Этим они прежде всего отличаются от функций, рассмотренных в предыдущих разделах, поскольку в тех функциях реакция пользователя сводилась к нажатию какой-то из кнопок.

Функция **InputBox** объявлена в модуле **Dialogs** следующим образом:

```

extern PACKAGE AnsiString __fastcall InputBox(
    const AnsiString ACaption,
    const AnsiString APrompt,
    const AnsiString ADefault);

```

Она предлагает пользователю диалоговое окно (см. рис. 15.6) с заголовком **ACaption**, с предложением **APrompt** пользователю что-то написать и с окошком редактирования, в котором предварительно загружено начальное значение текста **ADefault**. Если пользователь нажмет в окне **OK**, то функция вернет введенную им строку текста. Если же пользователь в диалоге нажал **Cancel**, или нажал **Esc**, или закрыл окно системной кнопкой, то функция вернет строку **ADefault**, даже если перед этим пользователь что-то написал в окошке редактирования.

Например, оператор

```

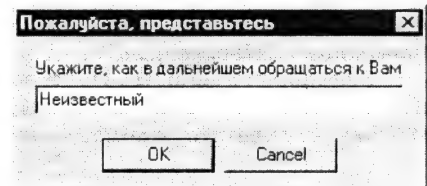
AnsiString Name = InputBox("Пожалуйста, представьтесь",
    "Укажите, как в дальнейшем обращаться к Вам",
    "Неизвестный");

```

отобразит окно, представленное на рис. 15.6, и вернет текст введенный пользователем, или строку «Неизвестный». Еще один пример использования **InputBox** приведен в разделе 15.7.2.5.

Рис. 15.6.

Окно, отображаемое функцией InputBox



Понять по возвращенному результату, написал ли пользователь какой-то текст, или отказался от ввода, можно, сравнив возвращенный результат со значением **ADefault**. Впрочем, результат останется неизменным и в случае, если пользователь ничего не написал в диалоге, но нажал кнопку ОК. Если надо достоверно знать, отказался ли пользователь от диалога, или нажал ОК, следует использовать похожую на **InputBox** функцию **InputQuery**:

```
extern PACKAGE bool __fastcall InputQuery(
    const AnsiString ACaption,
    const AnsiString APrompt,
    AnsiString &Value);
```

Смысл параметров **ACaption** и **APrompt** тот же, что в функции **InputBox**. Параметр **Value** — это строка текста в окошке редактирования. Вы можете присвоить ей начальное значение, а после вызова **InputQuery** в параметре **Value** будет находиться ответ пользователя.

Функция **InputQuery** возвращает **true** только в том случае, если пользователь вышел из диалога, нажав ОК. В остальных случаях (при нажатии Esc, при щелчке на системной кнопке окна или на кнопке Cancel) возвращается **false**, а значение параметра **Value** сохраняется тем, какое было до обращения к **InputQuery**.

Например, операторы:

```
AnsiString Name = "Неизвестный";
if(! InputQuery("Пожалуйста, представьтесь",
    "Укажите, как в дальнейшем обращаться к Вам", Name))
    ShowMessage("Вы не представились, господин Неизвестный");
else ShowMessage("Здравствуйте, господин "+Name+" !");
```

отобразят окно, представленное на рис. 15.6 и после окончания диалога выдадут сообщение, зависящее от того, нажал ли пользователь в диалоге ОК, или нет.

15.7.2.5 Функция **SelectDirectory**

Функция **SelectDirectory** предоставляет пользователю возможность указать в стандартном диалоге каталог. Функция объявлена в модуле **FileCtrl** и имеет две перегруженных формы. Первая форма:

```
extern PACKAGE bool __fastcall SelectDirectory(
    const AnsiString Caption,
    const WideString Root,
    AnsiString &Directory);
```

Функция вызывает стандартный диалог Windows для поиска каталога (папки) — см. примеры на рис. 15.7. Параметр **Caption** содержит строку, отображаемую в диалоге как указание пользователю (текст «Укажите каталог установки программы» на рис. 15.7. Параметр **Root** задает корневой каталог, внутри которого пользователь может выбирать подкаталоги. За пределы каталога **Root** пользователь выйти не может. При вызове **SelectDirectory** в примере рис. 15.7 а указано **Root = «d:\»**. Если указать вместо **Root** пустую строку или отсутствующий на компьютере каталог, то в диалоговом окне отобразится дерево всех папок (рис. 15.7 б) и пользователь имеет возможность выбрать на любом диске любой каталог.

Выходной параметр **Directory** содержит результат выбора пользователя. Функция возвращает **true**, если пользователь выбрал каталог и нажал ОК. Если пользователь нажал Отмена или закрыл каталог, не произведя выбора, то функция возвращает **false**.

Рассмотрим пример. Пусть вы делаете программу установки вашего приложения и хотите, чтобы пользователь указал каталог, в котором надо установить программу. Соответствующий диалог можно оформить следующим образом:

```

#include <FileCtrl.hpp>
...
AnsiString Dir;
...
if(SelectDirectory("Укажите каталог установки программы",
                  "", Dir))
    Dir = InputBox("Можете уточнить каталог",
                  "Программа расположится в каталоге:",
                  Dir);
else
{
    Application->MessageBox("Вы не указали каталог",
                           "Установка прерывана !",
                           MB_ICONSTOP);
    Application->Terminate();
}

```

В этом примере вызов функции **SelectDirectory** приводит к появлению окна, представленного на рис. 15.7 б, поскольку параметр **Root** указан пустой строкой. Если пользователь выбрал каталог и нажал ОК, то **SelectDirectory** возвращает **true**. В этом случае пользователю предлагается диалоговое окно, вызываемое функцией **InputBox** (см. раздел 15.7.2.4). Оно показано на рис. 15.8 а. В этом окне пользователь может, если хочет, уточнить каталог. Если же пользователь в окне

Рис. 15.7.

Диалоговое окно поиска каталога при заданном (а) и не заданном (б) значении **Root**

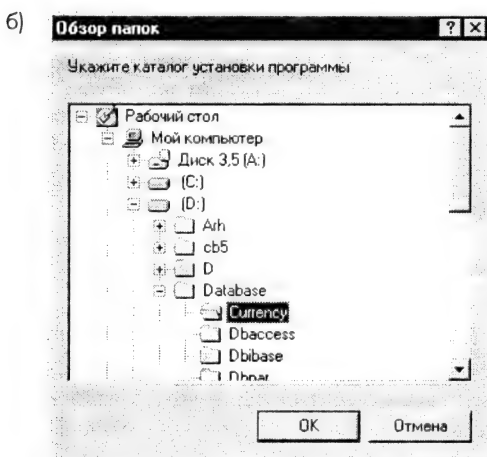
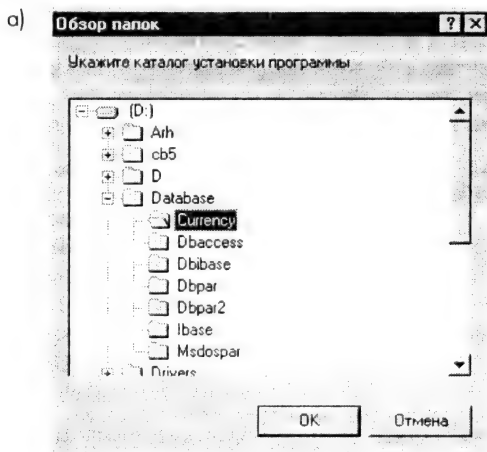


Рис. 15.8.

Продолжение диалога
выбора каталога

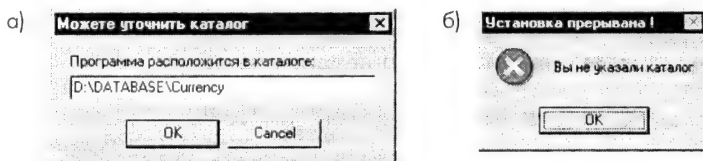


рис. 15.7 б не выбрал каталог, то функцией **Application.MessageBox** вызывается окно, показанное на рис. 15.8 б, и установка прерывается.

Функция **SelectDirectory** имеет еще вторую форму:

```
enum TSelectDirOpt { sdAllowCreate,
                    sdPerformCreate, sdPrompt };
typedef Set<TSelectDirOpt, sdAllowCreate, sdPrompt>
                    TSelectDirOpts;

extern PACKAGE bool __fastcall SelectDirectory(
                    AnsiString &Directory,
                    TSelectDirOpts Options,
                    int HelpCtx);
```

Эта форма вызова функции предоставляет более гибкий диалог (рис. 15.9). Возвращаемое значение по-прежнему указывает, выбрал ли пользователь каталог. Параметр **Directory**, как и раньше, содержит выбранный пользователем каталог. Если перед вызовом **SelectDirectory** задано начальное значение **Directory**, то именно этот каталог будет раскрыт в окне диалога в первый момент времени. Параметр **HelpCtx** является ссылкой на контекстную справку, содержащую подсказку по действиям пользователя. А параметр **Options** является множеством следующих опций:

sdAllowCreate	В диалоговом окне отображается окошко редактирования Directory Name (см. рис. 15.9), в котором пользователь может написать каталог, который отсутствует. Эта опция не создает сам каталог. Это задача приложения, которое прочтет имя каталога и при необходимости создаст его
sdPerformCreate	Применяется только в сочетании с sdAllowCreate и обеспечивает создание каталога, если указанный пользователем каталог отсутствует
sdPrompt	Применяется только в сочетании с sdAllowCreate . Если пользователь указал несуществующий каталог, ему предлагается вопрос, надо ли его создавать. Если пользователь ответил утвердительно (нажал OK) и опция sdPerformCreate включена в множество Options , то каталог будет создан. Если же опция sdPerformCreate не задана, то приложение должно само создать нужный каталог

Если множество **Options** пустое, то пользователь не может указать каталог, которого не существует.

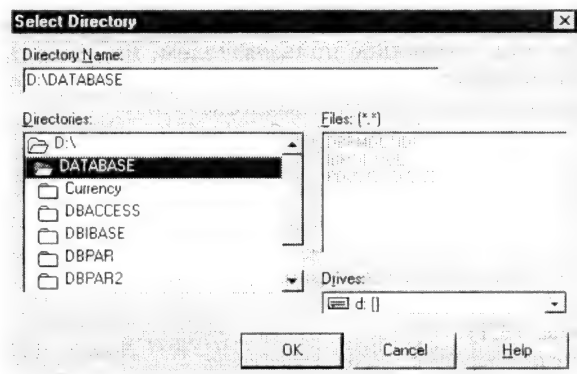
Если применить эту форму функции **SelectDirectory** в приведенном выше примере, то начало кода может иметь вид:

```
#include <FileCtrl.hpp>
...
AnsiString Dir = "d:\\";
...
if(SelectDirectory(Dir, TSelectDirOpts() << sdAllowCreate
                  << sdPerformCreate << sdPrompt, 0))
...

```

Рис. 15.9.

Вторая форма диалога выбора каталога

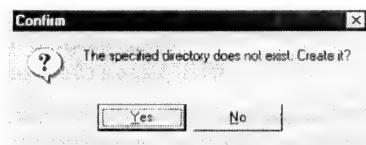


В этом коде задается начальное значение каталога и в параметр **Options** включены все опции: **sdAllowCreate**, **sdPerformCreate** и **sdPrompt**. Вызываемое диалоговое окно подобно приведенному на рис. 15.9, но не будет иметь кнопки **Help**, поскольку идентификатор контекстной справки задан равным нулю.

Если пользователь напишет в окошке редактирования **Directory Name** каталог, который отсутствует в дереве, ему будет предложено окно запроса, показанное на рис. 15.10. Если пользователь нажмет в нем **No**, то вернется в окно рис. 15.9. Если же он нажмет **Yes**, то отсутствующий каталог будет создан.

Рис. 15.10.

Запрос создания отсутствующего каталога



Как видно, вторая форма функции **SelectDirectory** дает дополнительную гибкость диалогу. Но она имеет существенный недостаток: окна рис. 15.7 и 15.9 содержат английские тексты. От окна запроса на создание каталога (рис. 15.10) безусловно лучше отказаться. Если допустимо создание нового каталога без дополнительного запроса, то следует задавать **Options** равным **[sdAllowCreate, sdPerformCreate]**. Если же запрос все-таки нужен, то лучше задать **Options** равным **[sdAllowCreate]**, а запрос и создание каталога обеспечить программно с помощью ранее рассмотренных средств. Так что от английского окна запроса избавиться несложно. Но с английскими надписями в основном диалоговом окне (рис. 15.9) сделать ничего невозможно.

15.7.2.6 Диалоги доступа к данным — функции **LoginDialog** и **LoginDialogEx**

Имеются две функции — **LoginDialog** и **LoginDialogEx**, вызывающие стандартные окна Windows, содержащие запрос имени и пароля пользователя для доступа к базам данных (см. рис. 15.11). Объявление функции **LoginDialog** в файле **Dblogdlg** имеет вид:

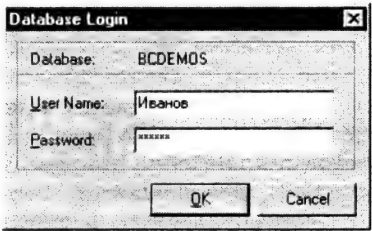
```
extern PACKAGE bool __fastcall LoginDialog(
    const AnsiString ADatabaseName,
    AnsiString &AUserName,
    AnsiString &APassword);
```

Параметр **ADatabaseName** является строкой, содержащей имя базы данных, с которой требуется соединиться. Параметр **AUserName** — строка, содержащая имя пользователя. При вызове функции **LoginDialog** в этот параметр можно записать имя пользователя по умолчанию и оно будет отображаться в диалоге в окошке **User**

Name (см. рис. 15.11). А по окончании диалога в параметре **AUserName** содержится имя, указанное пользователем, или имя по умолчанию, если пользователь его не изменял. Параметр **APassword** — строка, содержащая пароль, введенный пользователем в процессе диалога.

Функция **LoginDialog** возвращает **true**, если пользователь завершил диалог, щелкнув в нем на кнопке **OK**. В этом случае программа может попытаться открыть базу данных с указанными именем и паролем пользователя или предварительно проверить, зарегистрирован ли такой пользователь, правильный ли указан пароль и какой уровень доступа разрешен этому пользователю. Если пользователь прервал диалог, то возвращается **false**.

Рис. 15.11.
Запрос пароля функцией **LoginDialog**



Пример использования функции **LoginDialog:**

```
#include <Dblogdlg.hpp>
...
AnsiString DatabaseName = "BCDEMOS",
        UserName = "Иванов", Password;
if(! LoginDialog(DatabaseName, UserName, Password))
        ShowMessage("Вы не указали имя и пароль!");
else ...
```

В диалоговом окне, отображаемом функцией **LoginDialog**, пользователь может изменять имя, указываемое в окошке **User Name**. Другая функция **LoginDialogEx** — позволяет запретить изменение имени. Объявление этой функции имеет вид:

```
extern PACKAGE bool __fastcall LoginDialogEx(
        const AnsiString ADatabaseName,
        AnsiString &AUserName,
        AnsiString &APassword,
        bool NameReadOnly);
```

Отличие от функции **LoginDialog** заключается только в параметре **NameReadOnly**. Если положить его равным **true**, то изменение имени пользователем окажется невозможным. Само диалоговое окно при этом не будет отличаться от приведенного на рис. 15.11. Только окошко **User Name**, в котором по-прежнему будет отображаться указанное параметром **AUserName** имя, окажется недоступным для изменений.

Существенным недостатком рассмотренных функций **LoginDialog** и **LoginDialogEx** являются нерусифицированные диалоговые окна. Так что в большинстве приложений вряд ли стоит использовать эти функции. Окно, подобное рис. 15.11, очень легко создать самому, причем с русскими надписями.

15.7.3 Функции воспроизведения звуков

Функция	Синтаксис / описание	Файл
Beep	extern PACKAGE void Beep(void) Функция C++Builder , воспроизводит стандартный звуковой сигнал	SysUtils.hpp

Функция	Синтаксис / описание	Файл
Beep	BOOL Beep(DWORD dwFreq, DWORD dwDuration); Функция API Windows, только для Windows NT, воспроизводит звуковой сигнал с частотой dwFreq Герц и длительностью dwDuration миллисекунд	-
MessageBeep	BOOL MessageBeep(UINT uType); Функция API Windows, воспроизводит звуковой сигнал типа uType	-
PlaySound	BOOL PlaySound(LPCSTR pszSound, HMODULE hmod, DWORD fdwSound) Функция API Windows, воспроизводит звук указанного волнового файла, или звука системного события, или звука из ресурса	mmsystem.hpp

Комментарии

Функция C++Builder **Beep** воспроизводит стандартный звуковой сигнал, вызывая функцию **MessageBeep** API Windows с нулевым параметром. При этом воспроизводится стандартный звуковой сигнал, установленный в Windows, если компьютер имеет звуковую карту и стандартный сигнал задан (он устанавливается в «Панели управления» после щелчка на пиктограмме Звук). Если звуковой карты нет или стандартный сигнал не установлен, звук воспроизводится через динамик компьютера.

Воспроизведение асинхронное, т.е. приложение продолжает выполняться во время воспроизведения звука.

Функция **Beep** API Windows, примененная в Windows NT, синхронно воспроизводит звук простого тона через динамик и не возвращается до окончания звука. В Windows NT параметр **dwFreq** задает частоту звука в герцах. Он может иметь значения в диапазоне от 37 до 32,767 (от 0x25 до 0x7FFF). Параметр **dwDuration** устанавливает длительность звука в миллисекундах.

Воспроизведение синхронное: функция не возвращается до окончания воспроизведения звука.

Все сказанное относится только к Windows NT. В Windows 95 и 98 параметры игнорируются и функция становится подобной функции **Beep** C++Builder. Отличие этих функций остается только в том, что **Beep** C++Builder ничего не возвращает, а **Beep** API Windows при успешном выполнении возвращает ненулевое значение. При аварийном завершении она возвращает нуль. Тогда более развернутую информацию об ошибке можно получить вызовом функции **GetLastError** (см. раздел 15.7.5).

Компилятор автоматически разбирается, какая именно из функций **Beep** использована в программе, по наличию или отсутствию параметров.

Функция **MessageBeep** воспроизводит звуковой сигнал указанного типа. Звуки, соответствующие различным типам сигналов, хранятся в реестре в разделе [sounds] и устанавливаются пользователем с помощью программы «Панель управления» щелчком на пиктограмме Звук.

Целый без знака параметр **uType** функции **MessageBeep** определяет воспроизводимый звук. Для него predefinedены следующие константы:

Значение	Звук
0xFFFFFFFF	Стандартный звук через динамик
MB_ICONASTERISK	Звездочка

Значение	Звук
MB_ICONEXCLAMATION	Восклицание
MB_ICONHAND	Критическая ошибка
MB_ICONQUESTION	Вопрос
MB_OK	Стандартный звук

При успешном завершении функция возвращает ненулевое значение (**true**). Если функция вернула нулевое значение, то получить информацию об ошибке можно с помощью вызова **GetLastError** (см. раздел 15.7.5).

После инициализации воспроизведения звука функция **MessageBeep** возвращает управление в точку вызова и воспроизведение звука производится асинхронно.

Если функция **MessageBeep** не нашла указанный тип звука, она пытается воспроизвести стандартный звук. Если и он не установлен или если компьютер не снабжен звуковой картой, то звук воспроизводится через динамик компьютера.

Функция **PlaySound** API Windows воспроизводит звук указанного волнового файла, или звука системного события, или звука из ресурса.

Параметр **pszSound** представляет собой строку с нулевым символом в конце и определяет воспроизводимый звук. В зависимости от значений флага **fdwSound** (**SND_FILENAME**, **SND_ALIAS** или **SND_RESOURCE**) параметр **pszSound** может определять имя волнового файла, псевдоним системного события или идентификатор ресурса. Если и один из этих флагов не указан, функция ищет в реестре Windows или в файле WIN.INI указанное имя звука. Если звук найден, то он воспроизводится. Если звук не найден, то параметр **pszSound** интерпретируется как имя файла.

Звук, указанный параметром **pszSound**, должен помещаться в доступную память и должен подходить для установленного драйвера устройства воспроизведения волновых файлов. Функция **PlaySound** ищет файл звука в следующих каталогах: текущем, каталоге Windows, системном каталоге Windows, каталогах, перечисленных в переменной среды PATH, в списке каталогов, предоставляемых сетью. Более подробно последовательность поиска в каталогах рассмотрена в документации по функции **OpenFile**.

Если указанный звук не находится, функция **PlaySound** воспроизводит системный звук по умолчанию. Если функция не может найти и его, то воспроизведения не будет, а вернется значение **false**.

Если параметр **pszSound** задан равным 0, то воспроизведение любого волнового файла прерывается. Для прерывания воспроизведения звука, не связанного с волновым файлом, надо указывать **SND_PURGE** в параметре **fdwSound**.

Параметр **hmod** используется только при параметре **fdwSound** равном **SND_RESOURCE**. В этом случае **hmod** является дескриптором выполняемого файла, содержащего ресурс, который должен загружаться. В противном случае значение **hmod** задается равным 0.

Параметр **fdwSound** задает флаги воспроизведения звука. Флаги могут комбинироваться друг с другом операцией ИЛИ «|». Возможны следующие значения флагов:

SND_ALIAS

Параметр **pszSound** определяет псевдоним системного события в реестре Windows или в файле WIN.INI. Нельзя использовать совместно с **SND_FILENAME** и **SND_RESOURCE**

SND_ALIAS_ID

Параметр **szSound** является предопределенным идентификатором звука

SND_APPLICATION	Звук воспроизводится с использованием установок приложения
SND_ASYNC	Звук воспроизводится асинхронно и функция PlaySound возвращается немедленно после начала воспроизведения. Чтобы прекратить асинхронное воспроизведение волнового файла, надо вызвать PlaySound с параметром pszSound , равным 0
SND_FILENAME	Параметр pszSound является именем файла
SND_LOOP	Воспроизведение звука постоянно повторяется, пока не вызовется PlaySound с параметром pszSound , равным 0. Одновременно надо указать флаг SND_ASYNC асинхронного воспроизведения звука
SND_MEMORY	Файл звука события загружен в память. В этом случае параметр pszSound должен указывать на образ звука в памяти
SND_NODEFAULT	Звук события, кроме звука по умолчанию. Если указанный звук не найден, PlaySound вернется, не воспроизводя звук по умолчанию
SND_NOSTOP	Если заданный звук не может быть воспроизведен, поскольку ресурсы, необходимые для воспроизведения, заняты воспроизведением другого звука, функция PlaySound немедленно вернет false , не воспроизводя заданного звука. Если данный флаг не указан, функция PlaySound пытается остановить воспроизведение другого звука, чтобы устройство могло быть использовано для воспроизведения нового звука
SND_NOWAIT	Если драйвер занят, функция сразу вернется без воспроизведения заданного звука
SND_PURGE	Останавливается воспроизведение любых звуков, вызванных в данной задаче. Если pszSound не 0, останавливаются все экземпляры указанного звука. Если pszSound равен 0, то останавливаются все звуки, связанные с данной задачей. Отдельно надо указать дескриптор для остановки событий SND_RESOURCE
SND_RESOURCE	Параметр pszSound является идентификатором ресурса. Параметр hmod должен указывать на источник ресурса
SND_SYNC	Синхронное воспроизведение звука события. Функция PlaySound возвращается только после окончания воспроизведения

Функция **PlaySound** при успешном выполнении возвращает **true**, в противном случае — **false**.

Примеры использования функции **PlaySound** приведены в главе 5 в разделе 5.2.1.

15.7.4 Некоторые вспомогательные функции C++ и C++Builder

Функция	Синтаксис / Описание	Файл
ARRAYSIZE	ARRAYSIZE(const void *a) Макрос возвращает число элементов массива a	sysdefs.h
bsearch	void *bsearch(const void *key, const void *base, size_t nelem, size_t width, int (_USERENTRY *fcmp) (const void *, const void *)) Выполняет двоичный поиск по ключу key в массиве (таблице) base из nelem элементов по width байт каждый с помощью функции fcmp ; возвращает адрес элемента или 0	stdlib.h
EXISTING- ARRAY	EXISTINGARRAY(const void *a) Макрос возвращает индекс последнего элемента массива a	sysdefs.h
getenv	char *getenv(const char *name) Возвращает или удаляет переменную окружения name	stdlib.h
GetLongHint	extern PACKAGE System::AnsiString GetLongHint(const System::AnsiString Hint) Возвращает вторую часть строки формата, используемого в свойствах компонентов Hint	Controls.hpp
GetShortHint	extern PACKAGE System::AnsiString GetShortHint(const System::AnsiString Hint) Возвращает первую часть строки формата, используемого в свойствах компонентов Hint	Controls.hpp
lfind	void *lfind(const void *key, const void *base, size_t *num, size_t width, int (_USERENTRY *fcmp) (const void *, const void *)) Выполняет линейный поиск по ключу key в массиве (таблице) base из num записей по width байт в каждый с помощью функции fcmp ; возвращает адрес элемента или 0	stdlib.h
lsearch	void *lsearch(const void *key, void *base, size_t *num, size_t width, int(_USERENTRY *fcmp) (const void *, const void *)) Выполняет линейный поиск по ключу key в массиве (таблице) base из num записей по width байт в каждый с помощью функции fcmp ; если элемент не найден, он добавляется в таблицу; возвращает адрес элемента	stdlib.h
OPENARRAY	OPENARRAY(type arg1, ..., type arg19) Макрос обеспечивает передачу в функцию открытого массива, содержащего до 19 элементов	sysdefs.h

Функция	Синтаксис / Описание	Файл
ParamCount	extern PACKAGE int __fastcall ParamCount(void); Возвращает число параметров командной строки	System.hpp
ParamStr	extern PACKAGE AnsiString __fastcall ParamStr(int Index); Возвращает параметр с индексом Index командной строки	System.hpp
putenv	int putenv(const char *name) Устанавливает переменную окружения name	stdlib.h
qsort	void qsort(void *base, size_t nelem, size_t width, int (_USERENTRY *fcmp) (const void *, const void *)) Выполняет быструю сортировку в массиве (таблице) base из nelem элементов по width байт каждый с помощью функции fcmp	stdlib.h
ShortCut	extern PACKAGE TShortCut ShortCut(Word Key, Classes::TShiftState Shift) Создает структуру, используемую для задания комбинации «горячих» клавиш Key и Shift разделу меню	Menus.hpp
ShortCutToText	extern PACKAGE System::AnsiString ShortCutToText(TShortCut ShortCut) Преобразует структуру ShortCut , содержащую комбинацию «горячих» клавиш раздела меню, в строку текста	Menus.hpp
swab	void swab(char *from, char *to, int nbytes) Копирует nbytes байтов строки from в строку to , меняя местами каждую пару смежных байтов	stdlib.h
TextToShortCut	extern PACKAGE TShortCut TextToShortCut(System::AnsiString Text) Создает из строки текста Text структуру, используемую для задания комбинации «горячих» клавиш разделу меню	Menus.hpp
va_arg	type va_arg(va_list ap, type) Макрос возвращает текущий аргумент списка переменной длины типа type и переводит указатель ap на следующий аргумент; предварительно указатель должен быть установлен с помощью va_start или va_arg	stdarg.h
va_end	void va_end(va_list ap) Макрос обеспечивает завершение передачи в функцию списка аргументов произвольной длины, обработанного макросами va_start и va_arg	stdarg.h
va_start	void va_start(va_list ap, lastfix) Макрос устанавливает ap на первую переменную, передаваемую в функции, использующие списки аргументов произвольной длины; lastfix — последний переданный в функцию обязательный аргумент	stdarg.h

Комментарии

Макросы **va_start**, **va_arg** и **va_end** позволяют создавать функции, в которые передаются списки аргументов неопределенной длины. Иногда в функции требуется передавать некоторое число фиксированных параметров плюс неопределенное число дополнительных параметров. В этом случае заголовок функции имеет вид:

```
тип имя_функции(список_аргументов, ...)
```

В данном случае список аргументов включает в себя конечное число обязательных аргументов (этот список не может быть пустым), после которого ставится многоточие на месте неопределенного числа параметров. Для работы с этими параметрами в файле **stdarg.h** определены три макроса: **va_start**, **va_arg**, **va_end** и тип списка **va_list**.

Макрос **va_start** начинает работу со списком, устанавливая его указатель **ap** на первый передаваемый в функцию аргумент из списка с неопределенным числом аргументов. Параметр **lastfix** — это имя последнего из обязательных аргументов функции.

Макрос **va_arg** возвращает значение очередного аргумента из списка. Параметр **type** указывает тип аргумента. Перед вызовом **va_arg** значение **ap** должно быть установлено вызовом **va_start** или **va_arg**. Каждый вызов **va_arg** переводит указатель **ap** на следующий аргумент.

Макрос **va_end** завершает работу со списком, освобождая память. Он должен вызываться после того, как с помощью прочитан **va_arg** весь список аргументов. В противном случае могут быть непредсказуемые последствия.

Примеры использования всех этих макросов см в главе 12 в разделе 12.5.5.

Макросы **EXISTINGARRAY**, **ARRAYSIZE**, **OPENARRAY** используются при передаче в функции массивов. Описание способов работы с этими макросами см. в главе 13 в разделе 13.10.3. Примеры использования макросов приведены также в разделе 15.3.1.2.

Функции **bsearch**, **lfind**, **lsearch**, **qsort** предназначены для поиска и сортировки в массивах (таблицах). Во всех этих функциях параметр **base** указывает на начало массива, параметр **nelem** или **num** определяет число элементов, параметр **width** определяет число байтов, занимаемых элементом, а параметр **fcmp** указывает на функцию сравнения, которую вы должны определить и которая сигнализирует о результатах сравнения двух элементов, заданных своими указателями.

Функция **bsearch** осуществляет двоичный поиск (дихотомию) элемента, соответствующего ключу **key**. Подобный поиск самый быстрый, но он требует, чтобы элементы массива были расположены в порядке возрастания критерия поиска. Функция сравнения **fcmp** в данном случае получает как параметры два указателя: ***elem1** и ***elem2**. Функция должна провести сравнение значений, на которые они указывают, и вернуть результат сравнения:

< 0	при *elem1 < *elem2
== 0	при *elem1 == *elem2
> 0	при *elem1 > *elem2

Здесь знак < означает, что элемент ***elem1** расположен в массиве раньше элемента ***elem2**, знак > означает, что элемент ***elem1** расположен в массиве после элемента ***elem2**, а знак равенства означает, что элементы равны. Эта оговорка существенна, поскольку элементами могут быть не только числа, но и объекты любого типа, например, записи со множеством полей.

Функция **qsort** выполняет быструю сортировку массива по критерию, заданному функцией сравнения. Сама функция сравнения используется такая же, как описанная выше.

Ниже приведен пример использования функций **qsort** и **bsearch**. В примере объявлен массив **array** неупорядоченных целых чисел. Функция **fcmp** — это функция сравнения, одинаковая для **qsort** и **bsearch**. При щелчке на кнопке **Button1** массив сначала упорядочивается функцией **qsort**, а затем в нем ищется с помощью **bsearch** элемент, соответствующий указанному пользователем в окне **Edit1**.

```
#include <malloc.h>
#include <stdlib.h>
...
int array[] = {800,123,512,627,933,145};
...

int fcmp (const void *p1, const void *p2)
{ return (*(int*)p1 - *(int*)p2); }

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int key = StrToInt(Edit1->Text);
    int *elem;
    qsort(array, ARRAYSIZE(array), sizeof(int), fcmp);
    elem = (int *) bsearch (&key, array, ARRAYSIZE(array),
                           sizeof(int), fcmp);
    if(elem == 0) ShowMessage("Элемент " + IntToStr(key) +
                             " Не найден");
    else ShowMessage("Индекс элемента " + IntToStr(*elem) +
                     " равен " + IntToStr(elem - array));
}
```

Функции **lfind** и **lsearch** выполняют в массиве линейный поиск. Он медленнее, чем дихотомия, но может применяться к неупорядоченным массивам. Функции различаются тем, что **lsearch**, если элемент не обнаружен, добавляет его в конец массива. Функции поиска в **lfind** и **lsearch** отличаются от рассмотренных выше. Они должны возвращать нуль при совпадении элементов и ненулевое значение, если элементы различны.

Ниже приведен пример использования функции **lsearch**, похожий на рассмотренный ранее. В примере объявлен массив **array** неупорядоченных целых чисел. Функция **fcmp1** — это функция сравнения. При щелчке на кнопке **Button1** в массиве ищется элемент, соответствующий указанному пользователем в окне **Edit1**. Если элемент не найден, он добавляется в конец массива.

```
#include <malloc.h>
#include <stdlib.h>
int array[10] = {800,123,512,627,933,145};
unsigned Narray = 6;
...

int fcmp1 (const void *p1, const void *p2)
{ return (*(int*)p1 != *(int*)p2); }

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int key = StrToInt(Edit1->Text);
    int *elem;
    elem = (int *) lsearch (&key, array, &Narray,
                           sizeof(int), fcmp1);
    ShowMessage("Индекс элемента " + IntToStr(key) +
                " равен " + IntToStr(elem - array));
}
```

Функции **GetShortHint** и **GetLongHint** возвращают соответственно первую и вторую части строки формата

```
<текст первой части>|<текст второй части>
```


Строки такого вида, в частности, задаются в свойстве компонентов **Hint** (см. в главе 16 и в главе 4 раздел 4.1.9).

Функции **getenv** и **putenv** позволяют работать с переменными окружения. Переменные окружения представляют собой строки таблицы параметров окружения в виде **name=string\0**. Функция **getenv** ищет или удаляет указанную переменную окружения **name**. Если в функции **getenv** задать имя переменной окружения, она вернет указатель на строку, содержащую значение этой переменной. Например, оператор

```
Label1->Caption = getenv("PATH");
```

отображает в метке **Label1** содержание строки переменной **PATH**.

Имена переменных DOS и OS/2 должны записываться в верхнем регистре. Остальные переменные могут записываться как в верхнем, так и в нижнем регистрах.

Если переменная окружения с этим именем отсутствует, то возвращается **NULL**. Если в функции **getenv** задать параметр **name** в виде **"name="**, то переменная **name** будет удалена из окружения. Например, оператор

```
getenv("PATH=");
```

очистит переменную **PATH**.

Функция **putenv** устанавливает переменную окружения. Параметр **name** задается в виде **"name=string"**. Например:

```
putenv("PATH=c:\\temp");
```

Функции **Shortcut**, **ShortcutToText** и **TextToShortcut** используются для задания свойства **Shortcut** раздела меню, определяющего соответствующую этому разделу комбинацию «горячих» клавиш. Функция **Shortcut** упаковывает параметр **Key**, определяющий виртуальный код клавиши, и параметр **Shift**, задающий комбинацию вспомогательных клавиш типа Shift, Ctrl и Alt, в значение типа **TShortcut**, эквивалентное типу **Word**. Функция **TextToShortcut** создает аналогичное значение из строки текста. Функция **Shortcut** выполняется быстрее, но зато функцию **TextToShortcut** удобнее использовать в диалоге, когда комбинацию «горячих» клавиш задает пользователь с помощью окна редактирования.

Функция **ShortcutToText** позволяет получить текстовое описание значения **Shortcut** типа **TShortcut**. Эту функцию удобно использовать для вывода пользователю принятой в разделе меню комбинации «горячих» клавиш, если ему предоставляется возможность изменять эту комбинацию.

Рассмотрим примеры использования этих трех функций. Оператор

```
MOpen->Shortcut = Shortcut('O', TShiftState() << ssCtrl);
```

задает разделу меню с именем **MOpen** «горячие» клавиши Ctrl-O. Оператор

```
MOpen->Shortcut = Shortcut('O', TShiftState() << ssCtrl << ssAlt);
```

задает тому же разделу комбинацию Ctrl-Alt-O.

Еще один пример. Приведенные ниже процедуры обеспечивают задание пользователем комбинации «горячих» клавиш для раздела меню, названного в программе **Open**. Первая процедура с помощью **ShortcutToText** задает начальное значение текста в окне редактирования, равное исходной комбинации клавиш. А вторая — с помощью функции **TextToShortcut** изменяет комбинацию на заданную пользователем. Пользователь может задать, например, комбинацию Ctrl-O или как «^O», или как «Ctrl+O».

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Edit1->Text = ShortcutToText (MOpen->Shortcut);
}
//-----
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    MOpen->Shortcut = TextToShortcut(Edit1->Text);
}
```

Функции **ParamStr** и **ParamCount** позволяют работать с командной строкой. Функция **ParamStr** возвращает параметр командной строки с указанным индексом **Index**. Нулевым параметром командной строки является имя выполняемого файла приложения вместе с полным путем к нему. Таким образом, выражение **ParamStr(0)** вернет, например, строку «D:\\TEST\\PROJECT1.EXE», т.е. имя файла, приведенное к верхнему регистру. Из этого имени можно извлечь путь к выполняемому файлу. Это очень часто требуется, если программа использует какие-то другие файлы, расположенные в том же каталоге, в котором располагается выполняемый файл.

Если при запуске приложения в него через командную строку переданы какие-то параметры, то эти параметры могут быть прочитаны соответственно выражениями **ParamStr(1)**, **ParamStr(2)** и т.п. При этом регистр параметров будет тем, который использован при запуске программы. Так что при чтении параметров желательно программно приводить их к верхнему или нижнему регистрам.

Функция **ParamCount** возвращает число параметров, переданных через командную строку. Она позволяет организовывать циклы по параметрам командной строки. Например, код

```
for (int i=1;i<=ParamCount();i++)
    if (LowerCase(ParamStr(i)) == "-e")
        ...
```

обеспечивает выполнение некоторых действий (обозначены многоточием), если среди параметров командной строки встретится «-e» или «-E».

Следует отметить, что в файле объявлена переменная **CmdLine** типа (**char ***). Эта переменная содержит полный текст командной строки, в котором параметры отделены друг от друга пробелами. Регистр всех параметров в строке **CmdLine**, включая нулевой, соответствует тому, который использовался при запуске приложения на выполнение. Еще один альтернативный способ работы с командной строкой рассмотрен в главе 1 в разделе 1.5.3.

15.7.5 Некоторые вспомогательные функции API Windows

Функция	Синтаксис / описание
Close-Window	BOOL CloseWindow(HWND hWnd) Сворачивает, не уничтожая, окно, указанное дескриптором hWnd
Destroy-Window	BOOL DestroyWindow(HWND hWnd) Уничтожает окно, указанное дескриптором hWnd , и всех его потомков, освобождает отведенную память
Enable-Window	BOOL EnableWindow(HWND hWnd, BOOL bEnable) Делает доступным (при bEnable = true) или недоступным (при bEnable = false) окно, указанное дескриптором hWnd
Find-Window	HWND FindWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName) Возвращает дескриптор окна класса lpClassName с заголовком lpWindowName
GetLastError	DWORD GetLastError(VOID) Возвращает код последней ошибки
GetNext-Window	HWND GetNextWindow(HWND hWnd, UINT wCmd) Возвращает дескриптор следующего за hWnd или предыдущего окна в Z-последовательности

Функция	Синтаксис / описание
Get-Window	HWND GetWindow(HWND hWnd, UINT wCmd) Возвращает дескриптор окна, находящегося с указанным окном hWnd в указанном соотношении wCmd
GetWindowText	int GetWindowText(HWND hWnd, LPTSTR lpString, int nMaxCount) Копирует текст, связанный с окном или оконным элементом hWnd , в буфер lpString размера nMaxCount

Комментарии

Все приведенные в таблице функции API Windows используют для идентификации окна, к которому применяется функция, дескриптор **hWnd**. Для получение этого дескриптора можно использовать функцию **FindWindow**. Функция возвращает дескриптор окна верхнего уровня, класс и имя которого указаны; функция не ищет дочерние окна.

Параметр **lpClassName** указывает на строку с нулевым конечным символом, содержащую имя класса, или является атомом, указывающим на строку с именем класса. Если этот параметр атом, то это должен быть глобальный атом, созданный предварительно вызовом функции **GlobalAddAtom**. Атом является 16-битной величиной, которая должна помещаться в младшие разряды слова **lpClassName**, а старшие разряды должны быть нулевыми.

Параметр **lpWindowName** указывает на строку с нулевым конечным символом, содержащую имя окна (это свойство **Caption** формы, отображаемое в строке заголовка окна). Если этот параметр равен **NULL**, то считается, что под критерий поиска подходит любое окно указанного класса.

Если поиск прошел успешно, то функция **FindWindow** возвращает дескриптор окна, имеющего указанное имя класса и имя окна. В противном случае возвращается **NULL**.

Другой способ найти дескриптор окна — воспользоваться функцией **GetNextWindow**. Она определяет дескриптор следующего или предыдущего окна в Z-последовательности. Параметр **hWnd** — дескриптор окна, от которого начинается отсчет. Параметр **wCmd** определяет направление поиска. Если **wCmd = GW_HWNDNEXT**, то ищется следующее окно, находящееся ниже. Если **wCmd = GW_HWNDPREV**, то ищется предыдущее окно, находящееся выше.

Следующее окно в Z-последовательности — это то, которое вызывалось из указанного или к которому пользователь обращался после создания указанного окна. Если указано дочернее окно, то поиск ведется среди дочерних окон.

Если искомое окно найдено, то возвращается его дескриптор. Если следующего или предыдущего окна нет (в зависимости от значения **wCmd**), то возвращается 0. Развернутую информацию об ошибке можно получить вызовом функции **GetLastError**.

Примеры использования функций **FindWindow** и **GetNextWindow** см. в главе 6 в разделе 6.2.1.

Использование функции **GetNextWindow** дает тот же самый результат, что и вызов функции **GetWindow** со значениями параметра **GW_HWNDNEXT** или **GW_HWNDPREV**. Функция **GetWindow** обладает более широкими возможностями, позволяя определять дескриптор окна, находящегося с указанным окном **hWnd** в указанном соотношении (по Z-последовательности или по последовательности владельцев). Параметр этой функции **uCmd** определяет соотношения родства и может принимать значения:

GW_CHILD	Определяется дескриптор дочернего окна на вершине Z-последовательности, если указано родительское окно. В противном случае возвращается дескриптор NULL . Проверяются только дочерние окна указанного окна, но не его потомков
GW_HWNDFIRST	Определяется дескриптор окна того же типа, находящегося вверху Z-последовательности
GW_HWNDLAST	Определяется дескриптор окна того же типа, находящегося внизу Z-последовательности
GW_HWNDNEXT	Определяется дескриптор следующего окна в Z-последовательности
GW_HWNDPREV	Определяется дескриптор предыдущего окна в Z-последовательности
GW_OWNER	Определяется дескриптор окна, являющегося владельцем указанного

Если окно найдено, то возвращается его дескриптор. Если требуемого окна не существует, то возвращается 0. Развернутую информацию об ошибке можно получить вызовом функции **GetLastError**.

Функция **GetWindowText** копирует текст, связанный с указанным окном (отображаемый в его полосе заголовка) или оконным элементом с дескриптором **hWnd**, в буфер **lpString** размера **nMaxCount**. Если число символов в тексте превышает эту величину, текст усекается.

Функция посылает указанному окну или элементу, указанному в ее вызове, сообщение Windows **WM_GETTEXT**. Она не может воспринять текст окна редактирования из другого приложения.

Если функция выполнена успешно, она возвращает число скопированных символов, исключая завершающий нулевой символ. Если окно не имеет полосы заголовка или текст заголовка отсутствует, или при неверном дескрипторе возвращается нуль. Развернутую информацию об ошибке можно получить вызовом функции **GetLastError**.

Пример использования функции **GetWindowText** см. в главе 6 в разделе 6.2.1.

Мы рассмотрели возможности определения дескриптора окна. Но если окно, к которому обращается та или иная функция — это то окно, в модуле которого содержится ее вызов, то в качестве дескриптора можно подставлять **Handle**. Например, оператор

```
CloseWindow(Handle);
```

сворачивает данное окно.

При успешном выполнении большинство описанных функций возвращают ненулевое значение. При аварийном завершении возвращается нуль. Тогда более развернутую информацию об ошибке можно получить вызовом функции **GetLastError**.

Функция **GetLastError** возвращает значение кода последней ошибки данного потока. Эти коды индивидуальны для каждого потока и другие потоки их не изменяют.

Функция **GetLastError** должна вызываться сразу после возврата функции, ошибку которой вы хотите проверить. Это связано с тем, что некоторые функции вызывают при своем успешном завершении **SetLastError(0)**, что уничтожает код ошибки.

Большинство функций API Win32 устанавливают код последней ошибки при аварийном завершении, хотя некоторые устанавливают этот код в случае успешного выполнения. Обычно функции задают такие коды, как **FALSE**, **NULL**, **0xFFFFFFFF** или **-1**.

Код ошибки — 32-битовое значение. Наиболее значимый бит 31. Бит 29 зарезервирован для кода, определяемого приложением. В системных кодах этот бит не используется. Таким образом, этот бит показывает, что код был определен приложением. Это гарантирует отсутствие пересечения между вашими и системными кодами.

Чтобы получить строку сообщения об ошибке по коду, используйте функцию **FormatMessage**. Полный список кодов ошибок имеется в заголовочном файле **WINNT.H** в **Win32 SDK**.

Функция **EnableWindow** делает указанное окно доступным или недоступным для любых действий пользователя с помощью мыши или клавиатуры. Если окно недоступно, то предусмотренные в нем процессы продолжают, но пользователь не может воздействовать на это окно. Функция возвращает нуль, если окно было до этого доступным, и ненулевое значение, если оно не было доступным.

15.8 Сообщения Windows

15.8.1 Некоторые функции, константы и типы API Windows, используемые при работе с сообщениями

Работа с сообщениями Windows рассмотрена в главе 6 в разделе 6.3. Ниже приводятся справочные сведения по функциям, которые использовались в главе 6.

15.8.1.1 Функция **PostMessage**

Функция помещает указанное в ней сообщение окну или множеству окон в очередь сообщений потока, создавшего эти окна, и возвращается, не дожидаясь окончания обработки этого сообщения

Объявление

```
BOOL PostMessage(  
    HWND hWnd,      // дескриптор окна,  
                   // которому передается сообщение  
    UINT Msg,       // сообщение  
    WPARAM wParam,  // первый параметр сообщения  
    LPARAM lParam    // второй параметр сообщения  
);
```

Параметры

Параметр **hWnd** — дескриптор окна, которому передается сообщение. Если этот параметр равен **HWND_BROADCAST**, то сообщение передается всем окнам верхнего уровня в системе, включая недоступные, невидимые, перекрытые другими и всплывающие, за исключением дочерних окон. Если этот параметр **NULL**, то сообщение ставится в очередь сообщений (если она есть) текущего процесса.

Параметр **Msg** определяет передаваемое сообщение. Параметры **wParam** и **lParam** могут содержать дополнительную информацию.

Возвращаемое значение

Функция возвращает ненулевое значение при успешном завершении и нуль при аварийном завершении. В этом случае причину ошибки можно установить вызовом функции **GetLastError**.

Описание

Функция **PostMessage** ставит указанное в ней сообщение окну или множеству окон в очередь сообщений потока, создавшего эти окна, и возвращается, не дожидаясь окончания обработки этого сообщения. Этим функция **PostMessage** отличается от функции **SendMessage**, которая ждет окончания обработки сообщения и на это время блокирует приложение, пославшее сообщение. Сообщения в дальнейшем изымаются из очереди функциями **GetMessage** и **PeekMessage**.

Если вы посылаете сообщение в диапазоне ниже **WM_USER** асинхронными функциями **PostMessage**, **SendNotifyMessage** или **SendMessageCallback**, надо быть уверенным, что параметры сообщения не включают указателей. В противном случае из-за немедленного возврата функции может оказаться, что к моменту, когда поток начнет обрабатывать сообщение, приложение, которое его послало, окажется уже удаленным из памяти.

15.8.1.2 Функция **SendMessage**

Функция посылает указанное в ней сообщение окну или множеству окон и не возвращается, пока это сообщение обрабатывается

Объявление

```
LRESULT SendMessage(  
    HWND hWnd,          // дескриптор окна,  
                        // которому передается сообщение  
    UINT Msg,           // сообщение  
    WPARAM wParam,      // первый параметр сообщения  
    LPARAM lParam       // второй параметр сообщения  
);
```

Параметры

Параметр **hWnd** — дескриптор окна, которому передается сообщение. Если этот параметр равен **HWND_BROADCAST**, то сообщение передается всем окнам верхнего уровня в системе, включая недоступные, невидимые, перекрытые другими и всплывающие, за исключением дочерних окон.

Параметр **Msg** определяет передаваемое сообщение. Параметры **wParam** и **lParam** могут содержать дополнительную информацию.

Возвращаемое значение

Значение, возвращаемое функцией, зависит от вида сообщения.

Описание

Функция **SendMessage** посылает указанное в ней сообщение окну или всем окнам верхнего уровня в системе, включая недоступные и невидимые, кроме дочерних. Функция не возвращается, пока это сообщение обрабатывается. Таким образом, приложение, пославшее сообщение, блокируется на время его обработки. Этим функция **SendMessage** отличается от функции **PostMessage**, которая возвращается сразу после передачи сообщения.

Приложения, использующие **hWnd = HWND_BROADCAST** для связи между окнами разных приложений должны предварительно зарегистрировать уникальность своих сообщений функцией **RegisterWindowMessage**.

Примеры применения функции **SendMessage** рассмотрены в главе 6 в разделе 6.3.2.

15.8.1.3 Функция **RegisterWindowMessage**

Функция определяет новое окно сообщения с гарантированной уникальностью его в системе, которое может использоваться в функциях **SendMessage** и **PostMessage**.

Объявление

```
UINT RegisterWindowMessage(  
    LPCTSTR lpString      // адрес строки сообщения  
);
```

Параметры

lpString — указатель на строку с нулевым символом, содержащую регистрируемое сообщение.

Возвращаемое значение

Если регистрация прошла успешно, то возвращается идентификатор сообщения в диапазоне от 0xC000 до 0xFFFF. Если регистрация завершилась аварийно, то возвращается нулевое значение.

Описание

Функция используется для регистрации сообщений, предназначенных для связи между различными совместно работающими приложениями. Если два приложения регистрируют одну и ту же строку сообщения, то им возвращается одинаковый номер этого сообщения. Регистрация действительна до конца сеанса работы Windows.

Функцию **RegisterWindowMessage** следует использовать только в случаях, когда несколько приложений должны обрабатывать одно и то же сообщение. Для отправки собственных сообщений внутри данного класса оконных компонентов следует использовать любое целое в диапазоне от **WM_USER** до 0x7FFF.

15.8.1.4 Функция Perform

Метод класса **TControl**, посылающий оконному компоненту указанное сообщение Windows

Объявление

```
int __fastcall Perform(Cardinal Msg, int WParam, int LParam);
```

Параметры

Msg — идентификатор сообщения, **WParam** и **LParam** — параметры сообщения.

Описание

Метод **Perform** отличается от функций API Windows, описанных в предыдущих разделах, прежде всего тем, что это метод компонентов C++Builder. Метод посылает сообщение тому оконному компоненту, к которому он применен. При этом **Perform** заполняет поля структуры типа **TMessage** значениями параметров **Msg**, **WParam**, **LParam** и задает нулевое значение полю результата. Затем эта структура передается на обработку функции, указанной в компоненте свойством **WindowProc**. Таким образом сообщение пересылается непосредственно окну, метод **Perform** которого используется. Например, оператор

```
Form2->Perform(WM_CLOSE, 0, 0);
```

передает сообщение **WM_CLOSE** форме **Form2**, закрывая окно формы.

15.8.1.5 Константа WM_USER

Константа используется приложениями для определения своих частных сообщений

Описание

Константа **WM_USER** используется для разграничения номеров сообщений, резервированных для Windows, и частных сообщений оконных компонентов. Все возможные номера сообщений разделены на пять диапазонов:

от 0 до WM_USER - 1	Номера сообщений, используемые Windows
от WM_USER до 0x7FFF	Номера частных сообщений внутри данного класса оконных компонентов
от 0x8000 до 0xBFFF	Номера, зарезервированные для будущего использования в Windows
от 0xC000 до 0xFFFF	Номера, соответствующие строкам сообщений, используемым для обмена между приложениями и зарегистрированным функцией RegisterWindowMessage
свыше 0xFFFF	Номера, зарезервированные для будущего использования в Windows

Номера второго диапазона от от **WM_USER** до **0x7FFF** могут использоваться для определения и послыки сообщений внутри данного класса оконных компонентов. Их нельзя использовать для определения сообщений, предназначенных для обмена между приложениями, поскольку некоторые предопределенные классы оконных компонентов (например, **TButton**, **TEdit**, **TListBox** и **TComboBox**) уже используют этот диапазон. Сообщения другим приложениям в этом диапазоне могут послыаться только в случае, если приложения спроектированы с учетом обмена данными сообщениями и одинаково понимают номера этих сообщений.

Методика объявления и использования своих собственных сообщений и применения константы **WM_USER** вы можете найти в главе 6 в разделе 6.3.4.

15.8.1.6 Тип TMessage

Является типом параметра, характеризующего сообщения Windows и передаваемого в метод **WndProc**

Модуль *Messages*

Объявление

Messages

```
struct TMessage
{
    Cardinal Msg;
    union
    {
        struct
        {
            Word WParamLo;
            Word WParamHi;
            Word LParamLo;
            Word LParamHi;
            Word ResultLo;
            Word ResultHi;
        };

        struct
        {
            int WParam;
            int LParam;
            int Result;
        };
    };
};
```

Описание

Тип **TMessage** представляет в **WndProc** и других процедурах параметры сообщений Windows.

15.8.2 Некоторые сообщения Windows

15.8.2.1 WM_ACTIVATE

Сообщение посылается, когда окно переводится в активное или неактивное состояние. Сначала посылается окну, переходящему в неактивное состояние, а потом — активируемому.

Определение

```
WM_ACTIVATE
    fActive = LOWORD(wParam);
```



```
fMinimized = (BOOL) HIWORD(wParam);
hwndPrevious = (HWND) lParam;
```

Параметры

fActive — показывает, как активируется или деактивируется окно. Возможные значения:

WA_ACTIVE	активируется не щелчком мыши (например, функцией SetActiveWindow или клавиатурой)
WA_CLICKACTIVE	активируется щелчком мыши
WA_INACTIVE	деактивируется

fMinimized — ненулевое значение показывает, что окно минимизировано.

hwndPrevious — дескриптор, который указывает на окно, из которого фокус переключился на данное окно, если оно активируется, или на окно, в которое передается управление, если данное окно деактивируется.

Возвращаемое значение

Если приложение обрабатывает это сообщение, оно должно возвращать нуль.

Действие по умолчанию

Если активируемое окно не свернуто, то оно получает фокус.

Примечания

Если окно активируется щелчком мыши, оно получает также сообщение **WM_MOUSEACTIVATE**.

15.8.2.2 WM_ACTIVATEAPP

Сообщение посылается при переходе активности от окна одного приложения к окну другого приложения. Сообщения посылаются обоим окнам.

Определение

```
WM_ACTIVATEAPP
fActive = (BOOL) wParam;
dwThreadID = (DWORD) lParam;
```

Параметры

fActive — значение **true** означает, что окно становится активным, а **false** — что окно теряет активность.

dwThreadID — указывает сторонний процесс, который теряет или приобретает активность.

Возвращаемое значение

Если приложение обрабатывает это сообщение, оно должно возвращать нуль.

15.8.2.3 WM_CANCELMODE

Сообщение посылается окну, имеющему фокус при отображении модальных форм — диалогов и сообщений об ошибках. Дает возможность окну закрыться и освобождает мышь.

Возвращаемое значение

Если приложение обрабатывает это сообщение, оно должно возвращать нуль.

Действие по умолчанию

Внутренний процесс завершается и мышь освобождается.

15.8.2.4 WM_CLOSE

Сигнализирует, что окно или приложение закрывается.

Определение

WM_CLOSE

Возвращаемое значение

Если приложение обрабатывает это сообщение, оно должно возвращать нуль.

Действие по умолчанию

Вызывается функция **DestroyWindow**, уничтожающая окно.

Примечания

Приложение при обработке этого сообщения может запросить пользователя о необходимости закрывать окно и вызвать функцию **DestroyWindow** только при положительном ответе.

15.8.2.5 WM_GETMINMAXINFO

Посылается перед изменением размеров или положения окна. Обработчик сообщения может использоваться для ограничения допустимых размеров и координат положения на экране.

Определение

WM_GETMINMAXINFO

lpmmi = (LPMINMAXINFO) lParam;

Параметры

Параметр **lpmmi** указывает на структуру типа **MINMAXINFO**, содержащую принятые по умолчанию пределы изменения размеров и координат положения окна. Описание этой структуры:

```
typedef struct tagMINMAXINFO {
    POINT ptReserved;
    POINT ptMaxSize;
    POINT ptMaxPosition;
    POINT ptMinTrackSize;
    POINT ptMaxTrackSize;
} MINMAXINFO;
```

Поля структуры означают следующее:

ptReserved	Зарезервировано и пока не используется
ptMaxSize	Поле типа Point определяет ширину (Point.x) и высоту (Point.y) развернутого окна
ptMaxPosition	Поле типа Point определяет положения левого (Point.x) и верхнего (Point.y) краев развернутого окна
ptMinTrackSize	Поле типа Point определяет минимальную ширину (Point.x) и минимальную высоту (Point.y) окна при изменении пользователем размеров его рамки
ptMaxTrackSize	Поле типа Point определяет максимальную ширину (Point.x) и максимальную высоту (Point.y) окна при изменении пользователем размеров его рамки

Возвращаемое значение

Если приложение обрабатывает это сообщение, оно должно вернуть 0.

15.8.2.6 WM_GETTEXT

Посылается, чтобы скопировать текст, связанный с окном, в указанный буфер.

Определение

```
WM_GETTEXT
wParam = (WPARAM) cchTextMax;
lParam = (LPARAM) lpszText;
```

Параметры

cchTextMax указывает минимальное число символов, которые должны быть скопированы, включая нулевой конечный символ.

lpszText указывает на буфер, принимающий текст.

Возвращаемое значение

Возвращает число скопированных символов.

Действие по умолчанию

Копируется текст, связанный с окном, в указанный буфер и возвращается число скопированных символов.

Примечания

Для всех окон редактирования текст — это содержимое окна. Для выпадающих списков текст — это выделенный текст. Для кнопок текст — это имя кнопки. Для остальных оконных компонентов текст — это заголовок окна.

Для копирования обогащенного текста, превышающего 64K, надо использовать сообщения **EM_STREAMOUT** или **EM_GETSELTEXT**.

15.8.2.7 WM_SETTEXT

Посылается, чтобы задать текст указанного окна.

Определение

```
WM_SETTEXT
wParam = 0; // не используется, должен равняться 0
lParam = (LPARAM) (LPCTSTR) lpsz;
```

Параметры

lpsz — указатель на строку текста окна с нулевым конечным символом.

Возвращаемое значение

Возвращает **true**, если текст установлен. В противном случае возвращает **false** (для окна редактирования), **LB_ERRSPACE** (для списка) или **CB_ERRSPACE** (для выпадающего списка) если не хватает места для размещения текста. Возвращает **CB_ERR**, если сообщение посылается выпадающему списку без окна редактирования.

Действие по умолчанию

Устанавливает и отображает текст окна.

Примечания

Для всех окон редактирования текст — это содержимое окна. Для выпадающих списков текст — это выделенный текст. Для кнопок текст — это имя кнопки. Для остальных оконных компонентов текст — это заголовок окна.

Сообщение не изменяет текущее выделение в списках. Чтобы выделялся элемент списка, соответствующий тексту, надо использовать сообщение **CB_SELECTSTRING**.

Глава 16

Свойства, методы, события, типы, классы

В этой главе слова и термины, выделенные подобным образом, означают, что вы можете найти в главе раздел, содержащий развернутые пояснения выделенных терминов.

Определения свойств, методов, типов и, особенно, структур в приведенном в данной главе тексте несколько изменены и упрощены по сравнению с документацией C++Builder, чтобы читателю проще было в них ориентироваться. Например, опущены ссылки на методы записи и чтения свойств, которые не представляют интереса для пользователя, кардинально сокращены объявления многих структур — в них оставлены только имена и типы полей, устранены многие промежуточные ссылки на типы.

Конечно, приведенные в главе сведения — это малая часть всех свойств, методов, событий, классов, имеющих в C++Builder. В частности, из-за ограничения на размер книги в главе отсутствуют сведения по классам, свойствам, методам, используемым при работе с базами данных. Эти сведения (сверх того, что рассказано в главах 9 и 10) вам придется почерпнуть из встроеной в C++Builder справки, или можете посмотреть в книгах [7] и [8], где они приводятся более подробно, чем в данной книге, или воспользоваться русской справкой, которая, как я надеюсь, выйдет в серии «Все о C++Builder».

16.1 Свойства

Action

Определяет действие, связанное с данным управляющим элементом — разделом меню, кнопкой и др.

Класс *TControl*

Определение

```
__property Classes::TBasicAction* Action
```

Описание

Значение свойства **Action** выбирается во время проектирования из выпадающего списка предусмотренных действий в Инспекторе Объектов. Этот список формируется в процессе проектирования размещением на форме компонента **ActionList** и заданием его свойств.

Подробнее о диспетчеризации действий см. в разделе 3.9.1 главы 3.

Align

Определяет способ выравнивания компонента внутри контейнера (родительского компонента)

Класс *TControl*

Определение

```
enum TAlign {alNone, alTop, alBottom, alLeft, alRight, alClient};
```

__property TAlign Align

Описание

Свойство **Align** определяет, остается ли компонент неизменным при изменении размеров содержащей его формы, панели, другого компонента, или он изменяется, занимая всю доступную площадь, ее верхнюю, нижнюю, левую или правую часть.

Возможные значения свойства:

Значение	Описание
alNone	Компонент остается там, где он размещен во время проектирования. Размеры его не изменяются. Это значение Align по умолчанию.
alTop	Компонент занимает всю верхнюю часть контейнера и во время выполнения приложения его ширина изменяется при изменении ширины контейнера. Высота компонента остается неизменной.
alBottom	Компонент занимает всю нижнюю часть контейнера и во время выполнения приложения его ширина изменяется при изменении ширины контейнера. Высота компонента остается неизменной.
alLeft	Компонент занимает всю левую часть контейнера и во время выполнения приложения его высота изменяется при изменении высоты контейнера. Ширина компонента остается неизменной.
alRight	Компонент занимает всю правую часть контейнера и во время выполнения приложения его высота изменяется при изменении высоты контейнера. Ширина компонента остается неизменной.
alClient	Компонент занимает всю клиентскую область контейнера и во время выполнения приложения его размеры изменяются при изменении размеров контейнера. Если в контейнере часть клиентской области уже занята, компонент занимает всю ее оставшуюся часть.

Значение **Align** по умолчанию — **alNone**. В приложениях, в которых пользователь может изменять размер формы, а сама форма разбита панелями или другими компонентами на ряд областей, необходимо изменять это значение **Align**.

Если компонент имеет значение **Align**, равное **alClient**, то в процессе проектирования невозможно добраться до содержащего его контейнера и щелкнуть на нем, чтобы получить в Инспекторе Объектов его свойства и события. В этом случае возможны два решения: щелкнуть на компоненте и нажать клавишу Esc или осуществить выбор компонента-контейнера с помощью выпадающего списка в верхней части Инспектора Объектов.

Значения **Align alTop** и **alBottom** имеют приоритет перед **alLeft** и **alRight**. Поэтому, если вы, например, ввели на форму две панели, одной задали значение **alLeft**, а второй задаете значение **alTop**, то вторая панель вытеснит верхнюю часть первой панели, которая первоначально заняла всю левую часть клиентской области. Если это нежелательно, приходится вводить дополнительные панели, являющиеся контейнерами для других панелей.

Подробнее об использовании свойства **Align** и множество примеров его использования вы найдете в главе 4 в разделе 4.2.1.

Аnchors

Определяет привязку данного компонента к родительскому при изменении размеров последнего

Класс *TControl*

Определение

```
enum TAnchorKind { akLeft, akTop, akRight, akBottom };

typedef Set<TAnchorKind, akLeft, akBottom> TAnchors;

__property TAnchors Anchors
```

Описание

Свойство **Anchors** введено только начиная C++Builder 4. Оно определяет привязку данного компонента к родительскому при изменении размеров последнего. Свойство представляет собой множество типа **Set**, которое может содержать следующие элементы:

akTop	Компонент привязан к верхнему краю родительского
akLeft	Компонент привязан к левому краю родительского
akRight	Компонент привязан к правому краю родительского
akBottom	Компонент привязан к нижнему краю родительского

Если в множестве **Anchors** присутствуют привязки к противоположным сторонам родительского компонента, то при изменении родительского компонента происходит растяжение или сжатие данного компонента, поскольку расстояния от сторон родительского компонента выдерживаются. Сжатие может происходить вплоть до полного уничтожения изображения данного компонента. Для компонента **TPaintBox**, привязанного к противоположным сторонам родительского компонента, при изменении размеров родительского компонента изображение стирается и наступает событие **OnPaint**.

Примеры

1. `Button1->Anchors.Clear();`
`Button1->Anchors << akLeft << akBottom;`

Первый из этих операторов очищает множество **Anchors** кнопки **Button1**, удаляя из него первоначальные установки. Второй оператор привязывает кнопку **Button1** к левому и нижнему краям окна. При изменении размеров окна кнопка будет перемещаться, сохраняя установленное расстояние от левого и нижнего краев формы.

2. Задание компоненту списку **TListBox** свойства **Anchors**, равного **[akLeft, akTop, akBottom]**, приведет к тому, что при изменении высоты окна будут поддерживаться постоянными расстояния верхнего и нижнего краев компонента соответственно от верхнего и нижнего краев окна. Таким образом, увеличивая высоту окна пользователь может увеличивать число строк, видимых в списке без прокрутки. При этом необходимые с точки зрения эстетики расстояния до краев окна будут поддерживаться автоматически.
3. `Button1->Anchors.Clear();`
`Button1->Anchors << akLeft << akBottom << akRight;`

В этом случае кнопка привязана к левому, нижнему и правому краям формы. При изменении высоты окна кнопка будет перемещаться синхронно с нижним краем окна. А при изменении горизонтального размера окна кнопка будет растягиваться или сжиматься по горизонтали, что, конечно, будет создавать ужасный зрительный эффект. При сжатии кнопка может сжаться до нуля, после чего ее изображение исчезнет.

AutoMerge

Определяет, должно ли главное меню вторичной формы объединяться с меню главной формы

Класс *TMainMenu*

Определение

`__property bool AutoMerge`

Описание

Если требуется, чтобы меню вторичных форм объединялись с меню главной формы, то в каждой такой вторичной форме надо установить **AutoMerge** в **true**. При этом свойство главной формы должно оставаться в **false**. Способ объединения меню определяется свойствами **GroupIndex** элементов меню **Items** типа **TMenuItem**.

В MDI приложениях объединение меню осуществляется автоматически независимо от значения свойства **AutoMerge**.

AutoSelect

Указывает, будет ли выделяться весь текст, когда элемент получает фокус

Класс *TCustomEdit*

Определение

`__property bool AutoSelect`

Описание

Свойство **AutoSelect**, установленное в **true**, показывает, что при получении элементом фокуса весь текст окажется выделенным. Это свойство относится только к элементам редактирования одной строки текста.

Свойство **AutoSelect** имеет смысл устанавливать в **true**, если по условиям работы пользователь будет скорее заменять текст, имеющийся в окне, чем дополнять его.

AutoSize

Определяет, должны ли размеры компонента автоматически адаптироваться к размерам его текста или изображения

Классы *TCustomEdit*, *TCustomLabel*, *TImage* и др.

Определение

`__property bool AutoSize`

Описание

При свойстве **AutoSize**, установленном в **false**, размеры компонента фиксированы. При **AutoSize**, установленном в **true**, в классах-наследниках **TCustomLabel** (метках), в **TImage** и ряде других компонентов др. ширина и высота компонента автоматически адаптируется к размерам текста или изображения. В классах-наследниках **TCustomEdit** (окнах редактирования) ширина компонента остается фиксированной, а высота изменяется так, чтобы высота клиентской области соответствовала высоте текста. Например, высота элемента меняется при изменении шрифта или стиля бордюра.

Bitmap

Определяет внешний нестандартный шаблон размером 8 на 8 пикселей, который использует для заполнения кисть **Brush**

Класс *TBrush*

Определение

```
__property TBitmap* Bitmap
```

Описание

Свойство кисти **Bitmap** указывает на объект типа **TBitmap**, в который загружен шаблон размером 8 на 8 пикселей, используемый для заполнения кистью **Brush**.

Если для кисти задан шаблон **BitMap**, то заполнение производится именно этим шаблоном, независимо от значения свойства кисти **Style**. Шаблон **BitMap** может создаваться в процессе выполнения приложения или, например, загружаться из файла.

Если размер изображения превышает 8 на 8 пикселей, то в качестве шаблона будет использоваться его левая верхняя часть размером 8 на 8.

Изменение изображения в объекте **TBitmap** не влияет на шаблон, пока не произведено повторное присваивание свойству **Bitmap**.

После окончания работы с шаблоном объект **TBitmap** следует удалить из памяти, так как автоматически это не делается.

Пример

```
Graphics::TBitmap *MyBitmap = new Graphics::TBitmap;

try
{
    MyBitmap->LoadFromFile("MyBitmap.bmp");
    Image1->Canvas->Brush->Bitmap = MyBitmap;
    ...
}
__finally
{
    Image1->Canvas->Brush->Bitmap = NULL;
    delete MyBitmap;
}
```

В этом примере создается объект **MyBitmap** типа **TBitmap** и в него загружается битовая матрица из файла с именем «MyBitmap.bmp». Затем свойству **Image1->Canvas->Brush->Bitmap** присваивается указатель на этот объект. После этого загруженный шаблон можно использовать для заполнения фигур на канве **Image1**. В конце кода свойству **BitMap** присваивается значение **NULL**, после чего заполнение опять начинает определяться свойством **Style**. Затем объект **MyBitmap** уничтожается, чтобы освободить занимаемую им память.

BoundsRect

Определяет прямоугольник, описывающий компонент, в координатах содержащего его контейнера

Класс *TControl*

Определение

```
struct TRect
{
    int Left;
    int Top;
    int Right;
    int Bottom;
};

__property Windows::TRect BoundsRect
```


Описание

Свойство **BoundsRect** использует тип **TRect** и позволяет получить одновременно координаты пикселей всех четырех углов компонента. Иной способ получить те же координаты — использовать свойства компонента **Left** (левый край), **Top** (верхний край), **Width** (ширина), **Height** (высота) и соответствующие вычисления. Иначе говоря, при чтении данных эквивалентны следующие выражения:

Control->BoundsRect.Left	Control->Left
Control->BoundsRect.Top	Control->Top
Control->BoundsRect.Right	Control->Left + Control->Width
Control->BoundsRect.Bottom	Control->Top + Control->Height

Началом координат считается левый верхний угол родительского окна.

Отмеченная выше эквивалентность выражений для различных свойств справедлива только при чтении данных. Но присвоить целое значение, например, свойству **BoundsRect.Left** нельзя. Точнее, можно, но это не повлияет на размер компонента. Присвоить значение можно только всей структуре **BoundsRect**.

При задании значений **BoundsRect** удобно пользоваться функцией **Rect** (см. раздел 15.3.3 главы 15), принимающей координаты сторон прямоугольника и возвращающей структуру типа **TRect**.

Помимо перечисленных свойств, определяющих размеры компонента, имеются еще свойства **ClientWidth** и **ClientHeight**, определяющие размеры его клиентской области. Эти размеры равны или меньше размеров **Width** и **Height**.

Свойства, определяющие координаты компонента, полезны в задачах, требующих изменения размеров или перемещения компонентов.

Примеры

1. Пусть панель **Panel1** может менять свою длину при изменении пользователем размеров окна (например, имеет значение **Align = alTop**). И пусть в середине этой панели имеется метка **StaticText1** типа **StaticText**, которая при всех изменениях должна оставаться посередине, не изменяя своих размеров. Это можно осуществить, вставив в обработчик событий формы **OnResize** и **OnShow** оператор:

```
StaticText1->Left =
    (Panel1->BoundsRect.Left + Panel1->BoundsRect.Right -
     StaticText1->Width) / 2;
```

Впрочем, того же эффекта можно добиться и не прибегая к свойству **BoundsRect**, заменив приведенный оператор на следующий:

```
StaticText1->Left := Panel1->Left + (Panel1->Width -
    StaticText1->Width) / 2;
```

2. Пусть мы хотим, чтобы при нажатии некоторой кнопки окно текстового редактора **Memo1** перемещалось в новую заранее определенную позицию с новыми определенными размерами. Мы можем в нужной позиции разместить панель **Panel1** с нужными размерами, сделать ее невидимой, а в нужный момент выполнить всего один оператор:

```
Memo1->BoundsRect = Panel1->BoundsRect;
```

3. Пусть мы хотим взаимно поменять места расположения двух одинаковых по размерам панелей **Panel1** и **Panel2**. Это можно сделать следующими операторами:

```

TRect rec;
...
rec = Panel1->BoundsRect; // Запоминание позиции Panel1
Panel1->BoundsRect = Panel2->BoundsRect; // Перемещение Panel1
Panel2->BoundsRect = rec; // Перемещение Panel2

```

4. Пример задания **BoundsRect** с помощью функции **Rect**. Пусть окно текстового редактора **Memor1** расположено на родительской панели **Panel1**, размеры которой во время выполнения могут изменяться при изменении размеров формы. Надо, чтобы размер **Memor1** также изменялся, но по отношению к клиентской области **Panel1** оставял слева, внизу и справа зазор в 10 пикселей (для более приятного вида), а сверху — зазор 40 пикселей (для размещения заголовка окна). Это можно сделать, поместив в обработчик события **OnResize** панели **Panel1** оператор:

```

Memor1->BoundsRect = Rect(10,40,Panel1->ClientWidth-10,
                          Panel1->ClientHeight-10);

```

5. Примеры использования свойства **BoundsRect** приведены также в разделах **BringToFront** и **Visible**.

Break

Определяет, должен ли начинаться с данного раздела новый столбец разделов меню

Класс *TMenuItem*

Определение

```
enum TMenuBreak { mbNone, mbBreak, mbBarBreak };
```

```
__property TMenuBreak Break
```

Описание

Свойство **Break** используется в длинных меню, чтобы разбить список разделов на несколько столбцов. Возможные значения **Break**:

mbNone	Отсутствие разбиения меню. Это значение принято по умолчанию.
mbBar-Break	В меню вводится новый столбец разделов и данный раздел является верхним в новом столбце. Столбец отделяется от предыдущего полосой.
mbBreak	В меню вводится новый столбец разделов и данный раздел является верхним в новом столбце. Столбец отделяется от предыдущего пробелами.

Пример использования свойства **Break** приведен в разделе 3.6.1 главы 3.

Brush

Определяет цвет и стиль заполнения фона окна

Класс *TWinControl*

Доступ только для чтения

Определение

```
__property Graphics::TBrush* Brush
```

Описание

Свойство **Brush** (кисть) присуще многим оконным объектам, включая **Canvas**. Его можно читать, чтобы определить цвет и стиль заполнения фона окна. Это свойство только для чтения. Однако, атрибуты объекта **Brush** можно изменять, используя свойства **Color** и **Style**. Кроме того все свойства объекта могут быть изменены методом **Assign**.

Тип **TBrush** определяет свойства и методы объекта **Brush**. См. информацию о свойствах и методах, а также примеры в разделе **TBrush**.

Canvas

Поверхность (холст, канва) для рисования во многих компонентах

Классы *TForm, TImage, TBitMap, TPaintBox*

Доступ *только для чтения*

Определение

```
__property Graphics::TCanvas* Canvas
```

Описание

Свойство **Canvas** типа **TCanvas** используется для рисования пером **Pen** и кистью **Brush**, для модификации изображения, наложения друг на друга нескольких изображений.

В компоненте типа **TImage** канва может использоваться только в случае, если в свойство **Picture** загружена битовая матрица или ничего не загружено. Если в **Picture** находится объект типа, отличного от **TBitMap**, то при обращении к **Canvas** генерируется исключение **EInvalidOperation**. Если же в **Picture** находится битовая матрица, то **Canvas** можно использовать для редактирования изображения, например, для добавления в изображение надписей методом **TextOut**.

Capacity

Количество элементов массива указателей, которые могут храниться в объекте класса **TList**

Классы *TList, TStringList*

Определение

```
__property int Capacity
```

Описание

Свойство **Capacity** можно задавать, чтобы определить число указателей, которые могут храниться в объекте класса **TList**. Если при увеличении **Capacity** оказывается, что ресурсы памяти исчерпаны, генерируется исключение **EOutOfMemory**.

Свойство **Capacity** отличается от близкого ему свойства **Count**. **Capacity** показывает, сколько указателей может храниться в массиве, т.е. емкость списка, а **Count** показывает, сколько указателей в действительности хранится в массиве. Поэтому значение **Capacity** всегда не меньше значения **Count**.

При добавлении в массив нового элемента **Count** автоматически увеличивается на 1. Если при этом значение **Count** превышает значение **Capacity**, то **Capacity** тоже автоматически увеличивается, причем с запасом (обычно запас равен 3). Так что свойство **Capacity** вообще можно нигде в приложении не задавать. Смысл задания **Capacity** заключается только в том, чтобы предотвратить слишком частое перераспределение памяти при добавлении новых элементов списка.

При удалении элементов списка **Count** автоматически уменьшается, а значение **Capacity** остается неизменным. Таким образом, при сокращении длины списка получают излишние затраты памяти. В эти случаях целесообразно уменьшить **Capacity** до значения **Count** (если не планируется в ближайшее время нового увеличения числа элементов).

Если задать значение **Capacity** меньше, чем **Count**, генерируется исключение **EListError** с сообщением: «List capacity out of bounds [...]» («Емкость списка вне допустимых пределов»). В квадратных скобках указывается значение **Count**.

Очистка списка методом **Clear** сбрасывает **Capacity** на нуль.

Значение **Capacity** может быть увеличено методом **Expand**.

Пример

```
TList *List = new TList();
...
List->Clear();
List->Capacity = 5;
// Тут List->Count = 0 и List->Capacity = 5
...
for(int I = 1; I <= 5; I++)
    List->Add(...);
// Тут List->Count = 5 и List->Capacity = 5
...
for (int I = 0; I <= 2; I++)
    List->Delete(I);
// Тут List->Count = 2 и List->Capacity = 5
...
List->Capacity = List->Capacity - 3;
// Тут List->Count = 2 и List->Capacity = 2
...
```

Этот пример будет выполнять те же самые функции и без операторов

```
List->Capacity = 5;
```

и

```
List->Capacity = List->Capacity - 3;
```

Но первый из этих операторов экономит время при увеличении длины списка, а второй — сокращает затраты памяти.

Caption

Определяет строку текста, идентифицирующую компонент для пользователя

Класс *TControl*

Определение

```
__property System::AnsiString Caption
```

Описание

Свойство **Caption** связывает с компонентом некоторую строку текста, поясняющую его назначение. Чаще всего это надписи на кнопках, метках, панелях, тексты разделов меню и т.д.

По умолчанию **Caption** совпадает с именем компонента — свойством **Name** и изменяется при его изменении.

Для разделов меню и кнопок можно в свойстве **Caption** указать символы быстрого доступа. Для этого перед соответствующим символом надо поставить символ амперсанта — **&**. Следующий за амперсантом символ будет отображаться подчеркнутым и будет являться символом быстрого доступа: при выполнении приложения нажатие клавиши **Alt** + клавиши данного символа будет эквивалентно выбору соответствующего раздела меню или нажатию соответствующей кнопки.

Если в текст строки **Caption** надо ввести символ амперсанта, его надо повторить дважды: **&&**.

Для разделов меню задание значения **Caption**, равного символу «-», означает разделительную черту.

Поскольку **Caption** имеет тип **AnsiString**, то этому свойству можно непосредственно присваивать не только строковые значения, но и символы, целые и действительные числа.

Примеры

1. Примеры задания свойства **Caption** в процессе проектирования меню:

Задание	Отображение
&Файл	<u>Ф</u> айл
Фор&мат	Формат
Сохранить &как	Сохранить <u>к</u> ак

2. Пример использования метки для отображения результата ответа пользователя:

```
if (Edit1->Text = "1")
    Label1->Caption = "Ответ правильный";
else Label1->Caption = "Ответ ошибочный";
```

3. Примеры присваивания свойству **Caption** значений, отличных от строк:

```
Label1->Caption = 5;
Label2->Caption = 5.1;
Label3->Caption = 'A';
```

Charset

Свойство, определяющее набор символов шрифта

Класс *TFont*

Определение

```
typedef Byte TFontCharset;

__property TFontCharset Charset
```

Описание

Свойство **Charset** определяет набор символов шрифта — объекта типа **TFont**. Каждый вид шрифта, определяемый его именем, поддерживает один или более наборов символов. Какие именно значения **Charset** поддерживает тот или иной шрифт можно установить из документации на него или экспериментальным путем. Для шрифтов, поддерживающих несколько наборов символов, важно правильно задать **Charset**.

В C++Builder предопределено много констант, соответствующих стандартным наборам символов. Большинство из них, относящихся к японскому, корейскому, китайскому и другим языкам, вряд ли представляют интерес для наших читателей. Поэтому ниже приводится сокращенная таблица этих констант.

Константа	Значение	Описание
ANSI_CHARSET	0	Символы ANSI.
DEFAULT_CHARSET	1	Задается по умолчанию. Шрифт выбирается только по его имени Name и размеру Size . Если описанный шрифт недоступен в системе, то Windows заменит его другим шрифтом.
SYMBOL_CHARSET	2	Стандартный набор символов.

Константа	Значение	Описание
MAC_CHARSET	77	Символы Macintosh. Недоступны для NT 3.51.
GREEK_CHARSET	161	Греческие символы. Недоступны для NT 3.51.
RUSSIAN_CHARSET	204	Символы кириллицы. Недоступны для NT 3.51.
EASTEUROPE_CHARSET	238	Включает диакритические знаки (знаки, добавляемые к буквам и характеризующие их произношение) для восточно-европейских языков. Недоступны для NT 3.51.
OEM_CHARSET	255	Зависит от кодовой таблицы операционной системы.

По умолчанию в объекте типа **TFont** задается значение **Charset**, равное **DEFAULT_CHARSET**. Для имен шрифтов, принятых в C++Builder по умолчанию, это обычно нормальный вариант. Но в ряде случаев полезно для отображения русских текстов с другими шрифтами заменить это значение на **RUSSIAN_CHARSET**. Это позволит отобразить символы кириллицы для тех шрифтов, для которых при **DEFAULT_CHARSET** символы кириллицы не отображаются нормально.

Пример

См. пример 4 в разделе **Font**.

ClientHeight

Высота клиентской области в пикселях

Класс *TControl*

Определение

```
__property int ClientHeight
```

Описание

Свойство **ClientHeight** может использоваться для чтения или изменения высоты клиентской области. Чтение **ClientHeight** необходимо, чтобы изменять местоположение или размеры компонентов, расположенных в клиентской области, при изменении размеров компонента-контейнера (см. примеры в разделе **BoundsRect**).

Эквивалентно по значению свойству **ClientRect.Bottom** (см. **ClientRect**).

В классе **TControl** значение **ClientHeight** равно значению **Height**. Но в производных классах оно обычно меньше **Height**. Например, для формы значение **ClientHeight** может уменьшаться из-за бордюра, полосы заголовка, полосы меню, полосы прокрутки.

Пример

Пусть вы проектируете форму, содержащую панель **Panel1**, на которой размещается список **ListBox1**, причем во время выполнения высота этой формы может изменяться. Тогда обеспечить синхронное изменение размера окна списка можно, вставив в событие **OnResize** формы или этой панели оператор:

```
ListBox1->Height = Panel1->ClientHeight - ListBox1->Top - 20;
```

Более подробное рассмотрение этого примера имеется в разделе 4.2 главы 4.

ClientOrigin

Координаты положения на экране левого верхнего угла клиентской области компонента

Класс *TControl**Доступ только для чтения***Определение**

```
struct TPoint
{
    int x;
    int y;
};

__property Windows::TPoint ClientOrigin
```

Описание

Свойство **ClientOrigin** возвращает экранные координаты **x** и **y** левого верхнего угла клиентской области компонента. Горизонтальная координата **x** и вертикальная координата **y** хранятся в структуре типа **TPoint**. Началом координат считается левый верхний угол экрана.

Экранные координаты **ClientOrigin** для компонентов, не являющихся потомками класса **TWinControl** (т.е. не являющихся окнами), определяются как экранные координаты родительского компонента (компонента-контейнера), сложенные со значениями свойств **Left** и **Top**. Если компонент не имеет родителя, то при попытке чтения **ClientOrigin** генерируется исключение **EInvalidOperation**.

ClientRect

Определяет координаты углов клиентской области компонента

Класс *TControl**Доступ только для чтения***Определение**

```
struct TRect
{
    int Left;
    int Top;
    int Right;
    int Bottom;
};

__property Windows::TRect ClientRect
```

Описание

Свойство **ClientRect** возвращает структуру типа **TRect**, характеризующую прямоугольную клиентскую область компонента. При этом поля структуры **Top** и **Left** устанавливаются в нуль, а поля **Right** и **Bottom** определяются соответственно значениями свойств **ClientWidth** и **ClientHeight**. Так что **ClientRect** эквивалентна функции **Rect** (см. раздел 15.3.3 главы 15) вида **Rect(0, 0, ClientWidth, ClientHeight)**.

Координаты клиентской области могут задаваться как четырьмя целыми значениями координат, так и двумя точками, характеризующими левый верхний и правый нижний углы прямоугольника.

ClientWidth

Горизонтальный размер клиентской области в пикселях

Класс *TControl***Определение**

```
__property int ClientWidth
```

Описание

Свойство **ClientWidth** может использоваться для чтения или изменения горизонтального размера клиентской области. Чтение **ClientWidth** необходимо, чтобы изменять местоположение или размеры компонентов, расположенных в клиентской области, при изменении размеров компонента-контейнера (см. подобные примеры в разделе **BoundsRect**).

Эквивалентно по значению свойству **ClientRect.Right** (см. **ClientRect**).

В классе **TControl** значение **ClientWidth** равно значению **Width**. Но в производных классах оно обычно меньше **Width** за счет бордюра, полосы прокрутки и иных элементов оформления.

ClipRect

Определяет доступную область рисования на канве и область, подлежащую перерисовке при обработке события **OnPaint**.

Класс *TCanvas*

Доступ только для чтения

Определение

```
struct TRect
{
    int Left;
    int Top;
    int Right;
    int Bottom;
};

__property Windows::TRect ClipRect
```

Описание

Свойство канвы **ClipRect** определяет доступную область рисования на канве и область, нуждающуюся в перерисовке. Вне области **ClipRect** рисовать невозможно.

При обработке события формы **OnPaint** это свойство определяет ту часть канвы, вне которой перерисовка не требуется. Использование этого свойства позволяет сократить затраты времени на перерисовку.

Color

Цвет фона компонента, цвет текста объекта **TFont** и др.

Класс *TControl*

Определение

```
__property Graphics::TColor Color
```

Описание

Свойство **Color** определяет цвет фона компонента или цвет текста объекта **TFont**. Значение цвета может задаваться как значение, определяющее интенсивности красного, зеленого и синего цветов в формате RGB (см. раздел **TColor**), или равным одной из перечисленных ниже предопределенных в C++Builder констант.

Константа	Значение цвета
clBlack	Черный
clMaroon	Темно-бордовый
clGreen	Зеленый
clOlive	Оливково-зеленый

Константа	Значение цвета
clNavy	Темно-синий
clPurple	Пурпурный
clTeal	Морской воды
clGray	Серый
clSilver	Серебряный
clRed	Красный
clLime	Лимонно-зеленый
clBlue	Синий
clYellow	Желтый
clFuchsia	Сиреневый
clAqua	Голубой
clWhite	Белый
clBackground	Текущий цвет фона стола Windows
clScrollBar	Текущий цвет полос прокрутки
clActiveCaption	Текущий цвет фона полосы заголовка в активном окне
clInactiveCaption	Текущий цвет фона полосы заголовка в неактивном окне
clMenu	Текущий цвет фона меню
clWindow	Текущий цвет фона окон
clWindowFrame	Текущий цвет рамок окон
clMenuText	Текущий цвет текста меню
clWindowText	Текущий цвет текста окон
clCaptionText	Текущий цвет текста заголовка в активном окне
clActiveBorder	Текущий цвет бордюра активного окна
clInactiveBorder	Текущий цвет бордюра неактивного окна
clAppWorkSpace	Текущий цвет рабочей области приложений
clHighlight	Текущий цвет фона выделенного текста
clHightlightText	Текущий цвет выделенного текста
clBtnFace	Текущий цвет поверхности кнопок
clBtnShadow	Текущий цвет теней, отбрасываемых кнопками
clGrayText	Текущий цвет текста недоступных элементов
clBtnText	Текущий цвет текста кнопок
clInactiveCaption-Text	Текущий цвет текста заголовка в неактивном окне
clBtnHighlight	Текущий цвет выделенной кнопки
cl3DDkShadow	Цвет темных теней трехмерных элементов; только для Windows 95/98 или NT 4.0

Константа	Значение цвета
cl3DLight	Светлый цвет на краях освещенных трехмерных элементов; только для Windows 95/98 или NT 4.0
clInfoText	Цвет текста советов; только для Windows 95/98 или NT 4.0
clInfoBk	Цвет фона советов; только для Windows 95/98 или NT 4.0

Первая часть этих констант соответствует определенным цветам. А вторая часть определяется той схемой цветов, которую установил пользователь в Windows. Пользователь может менять эту схему с помощью «Панели Управления» Windows. Таким образом, эти цвета могут изменяться от системы к системе. Например, **clBtnFace** может соответствовать серому цвету в одной схеме и желто-коричневому в другой.

Как правило, лучше использовать в приложении эти системнозависимые цвета. Тогда ваше приложение не будет выбиваться из общей гаммы цветов, на которую настроился пользователь.

Если свойство **ParentColor** компонента установлено в **true** (это делается по умолчанию), то цвет данного компонента определяется свойством **Color** контейнера, содержащего данный компонент. Это позволяет автоматически согласовывать цвета компонентов, содержащихся в одной области окна. Как только вы задаете в Инспекторе Объектов или программно значение **Color**, свойство **ParentColor** переключается в **false**.

ComponentCount

Число компонентов, которыми владеет данный компонент

Класс *TComponent*

См. раздел **Components**.

ComponentIndex

Индекс компонента в списке компонентов владельца данного компонента

Класс *TComponent*

См. раздел **Components**.

Components

Массив компонентов, которыми владеет данный компонент

Класс *TComponent*

Определение

```
__property TComponent* Components[int Index]
```

Описание

Свойство **Components** содержит массив компонентов, которыми владеет данный компонент. Параметр **Index** позволяет сослаться на любой компонент с помощью его свойства **ComponentIndex**, определенного в классе **TComponent**. Индексы отсчитываются от 0, т.е. индекс первого компонента равен 0. Общее число компонентов, содержащихся в массиве **Components**, определяется свойством **ComponentCount**, определенным в классе **TComponent**. Значение **ComponentCount** на 1 меньше последнего индекса массива **Components**.

Свойство **Components** может использоваться вместе с **ComponentCount** в циклах, когда надо изменить какие-то свойства всех компонентов.

Примеры

1. В приведенном ниже примере все компоненты на данной форме, кроме компонента с именем **Button1**, смещаются вправо на 10 единиц.

```
for(int i = 0; i < ComponentCount; i++)
    if (Components[i]->Name != "Button1")
        ((TControl *)Components[i])->Left += 10;
```

2. Ниже приведен аналогичный пример, но используется свойство **Tag** и сдвигаются только компоненты, у которых **Tag = 1**.

```
for(int i = 0; i < ComponentCount; i++)
    if (Components[i]->Tag == 1)
        ((TControl *)Components[i])->Left += 10;
```

Constraints

Позволяет задавать ограничения на допустимые изменения размеров компонента при изменениях размеров окна приложения

Класс *TControl*

Определение

```
__property TSizeConstraints* Constraints
```

Описание

Свойство **Constraints** является объектом типа **TSizeConstraints**. Этот объект имеет четыре основных свойства: **MaxHeight**, **MaxWidth**, **MinHeight** и **MinWidth** — соответственно максимальная высота и ширина и минимальная высота и ширина компонента. Тип каждого из этих свойств — **TConstraintSize** целое без знака.

По умолчанию значения свойств **MaxHeight**, **MaxWidth**, **MinHeight** и **MinWidth** равны 0, что означает отсутствие ограничений. Но задание любому из этих свойств положительного значения приводит к соответствующему ограничению размера заданным числом пикселей.

Чтобы какие-то компоненты не исчезали из поля зрения при изменениях пользователем или программой размеров окна приложения, можно задать компонентам ограничения минимальной высоты и длины. Таким образом можно поддерживать нормальные пропорции отдельных частей окна. Можно задать ограничения на минимальные и максимальные размеры формы, т.е. всего окна. Например, если вы зададите для формы значения **MaxHeight = 500** и **MaxWidth = 500**, то пользователь не сможет сделать окно большим, чем квадрат 500 x 500. Причем это ограничение будет действовать даже если пользователь нажмет системную кнопку, разворачивающую окно на весь экран. Окно развернется, но его размеры не превысят заданных. Это иногда полезно делать, чтобы развернутое окно не заслонило какие-то другие нужные пользователю окна.

ControlCount

Число дочерних компонентов оконного элемента

Класс *TWinControl*

Доступ только для чтения

Определение

```
__property int ControlCount
```

Описание

Свойство **ControlCount** используется совместно со свойством **Controls** в итерациях по всем дочерним компонентам данного оконного элемента. Свойство **Controls**, с которым используется **ControlCount**, дает доступ к дочерним компонентам. Значение **ControlCount** всегда на 1 больше последнего индекса дочернего компонента, поскольку индексы считаются с 0.

См. примеры в разделе **Controls**.

Controls

Список дочерних компонентов оконного элемента

Класс *TWinControl*

Доступ *только для чтения*

Определение

```
__property TControl* Controls[int Index]
```

Описание

Свойство **Controls** является массивом всех дочерних компонентов данного оконного элемента. Дочерними являются те компоненты, которые расположены в клиентской области данного оконного элемента и в свойстве **Parent** которых указан как родитель данный элемент. Параметр **Index** определяет индекс соответствующего компонента. Индекс, начинающийся с 0, соответствует положению компонента в Z-последовательности данного родительского элемента.

Свойство **Controls** обычно используется в итеративных процедурах групповой обработки дочерних компонентов, когда на них неудобно ссылаться по имени. В подобных итеративных процедурах обычно используется также свойство **ControlCount**, определяющее число дочерних компонентов.

Надо четко представлять себе отличие свойства **Controls** от свойства **Components**. Свойство **Components** относится не к дочерним компонентам, а к тем, которыми владеет данный объект. В частности, всеми компонентами на форме владеет форма и они содержатся в ее списке **Components**.

Свойство **Controls** предназначено только для чтения. Оно изменяется (точнее меняются индексы компонентов) при изменении Z-последовательности.

Изменить Z-последовательность в процессе проектирования можно, выполнением команд **Bring To Front** или **Send To Back**. Первая из них пересылает выделенный компонент наверх, присваивая ему максимальный индекс, а вторая пересылает вниз, присваивая ему минимальный индекс (0 для неоконных компонентов и минимально возможный для оконных, поскольку всегда неоконные компоненты имеют индекс меньше оконных). Выполнить эти команды можно или из раздела меню **Edit**, или щелкнув правой кнопкой мыши и выбрав их из всплывающего меню.

Программно место компонента в Z-последовательности можно изменить методами **BringToFront** и **SendToBack**. На Z-последовательность влияют также методы **InsertControl** и **RemoveControl**, добавляющие и удаляющие дочерние компоненты, и изменение свойства компонентов **Parent**, меняющее родителя компонента.

Примеры

Пусть в приложении в классе формы определена некоторая функция

```
void __fastcall Func(TObject *Sender);
```

которая обрабатывает объект, передаваемый в нее через аргумент **Sender**. Это может быть какая-то процедура изменения размеров и места расположения, окрашивания, перестановок и т.д. Например, она может содержать оператор

```
((TControl *)Sender)->Left += 10;
```

сдвигающий объект на 10 пикселей вправо. Тогда обработать этой процедурой все дочерние компоненты, например, панели **Panel1** можно с помощью оператора:

```
for(int i = 0; i < Panel1->ControlCount; i++)
    Func(Panel1->Controls[i]);
```

Если надо обработать только какую-то группу компонентов с идущими по порядку индексами и известны их начальный и конечный индексы **Ind1** и **Ind2**, то соответствующий оператор может иметь вид

```
for(int i = Ind1; i <= Ind2; i++)
    Func(Panel1->Controls[i]);
```

Задать нужную последовательность индексов можно описанными выше способами в процессе проектирования или во время выполнения приложения.

Выяснить истинную последовательность индексов для той же панели **Panel1** можно, например, оператором:

```
for(int i = 0; i < Panel1->ControlCount; i++)
    ShowMessage("Ind = "+IntToStr(i) + ' ' +
        Panel1->Controls[i]->Name);
```

ControlState

Множество значений, характеризующих состояние компонента во время выполнения приложения

Класс *TControl*

Определение

```
enum Controls__7 { csLButtonDown, csClicked, csPalette,
    csReadingState, csAlignmentNeeded,
    csFocusing, csCreating, csPaintCopy,
    csCustomPaint, csDestroyingHandle,
    csDocking };

typedef Set<Controls__7, csLButtonDown, csDocking>
    TControlState;

__property TControlState ControlState
```

Описание

Свойство **ControlState** определяет различные условия, действующие на данный экземпляр компонента, например, щелчок мыши или необходимость выравнивания компонента. Свойство используется в основном при создании новых классов, производных от **TControl**. Свойство является множеством типа **Set** и может содержать следующие флаги:

Состояние	Пояснение
csLButtonDown	Левая кнопка мыши нажата, но еще не освобождена
csClicked	То же самое, что csLButtonDown , но только в том случае, если свойство компонента ControlStyle содержит флаг csClickEvents , означающее, что событие, связанное с нажатием кнопки, интерпретируется как щелчок
csPalette	Компонентом или одним из его родителей получено сообщение WM_PALETTECHANGED
csReadingState	Компонент читает свое состояние из потока
csAlignmentNeeded	Компонент должен осуществить выравнивание

Состояние	Пояснение
csFocusing	Приложение получило сообщение о переключении фокуса на данный компонент. Это не гарантирует, что компонент получит фокус, но позволяет предотвратить рекурсивные вызовы
csCreating	Создается данный компонент, или его владелец, или управляемый им компонент. Этот флаг очищается, когда создание компонента завершено
csPaintCopy	Компонент должен быть перерисован. Это состояние возможно, если свойство ControlStyle содержит флаг csReplicable
csCustomPaint	Компонент обрабатывает сообщения перерисовки
csDestroyingHandle	Окно компонента разрушается
csDocking	Компонент находится в процессе встраивания

Свойство **ControlState** характеризует не класс в целом, а конкретный объект класса.

ControlStyle

Множество значений, характеризующих стиль компонента

Класс *TControl*

Определение

```
enum Controls__8 { csAcceptsControls, csCaptureMouse,
  csDesignInteractive, csClickEvents,
  csFramed, csSetCaption, csOpaque,
  csDoubleClicks, csFixedWidth, csFixedHeight,
  csNoDesignVisible, csReplicable,
  csNoStdEvents, csDisplayDragImage,
  csReflector, csActionClient, csMenuEvents };
```

```
typedef Set<Controls__8, csAcceptsControls, csMenuEvents>
  TControlStyle;
```

```
__property TControlStyle ControlStyle
```

Описание

Свойство **ControlStyle** определяет различные атрибуты компонента, например, может ли он быть захвачен мышью или имеет ли он фиксированные размеры. Свойство используется в основном при создании новых классов, производных от **TControl**. Свойство является множеством типа **Set** и может содержать следующие флаги:

Флаг	Пояснение
csAcceptsControls	Компонент становится родителем любого компонента, перенесенного на него в процессе проектирования
csCaptureMouse	Компонент перехватывает события мыши после щелчка на нем
csDesignInteractive	Компонент устанавливает соответствие во время проектирования щелчка правой кнопки мыши щелчку левой кнопки для манипуляций с компонентом

Флаг	Пояснение
csClickEvents	Компонент получает сообщение о щелчке мыши и реагирует на него
csFramed	Компонент имеет объемную рамку
csSetCaption	Компонент должен изменять надпись на нем в соответствии со свойством Name , если только надпись не задана явным образом
csOpaque	Компонент полностью заполняет свою клиентскую область
csDoubleClicks	Компонент получает сообщение о двойном щелчке мыши и реагирует на него. Если флаг не установлен, то двойной щелчок интерпретируется как просто щелчок
csFixedWidth	Ширина компонента не меняется и не масштабируется
csFixedHeight	Высота компонента не меняется и не масштабируется
csNoDesign Visible	Компонент невидим во время проектирования
csReplicable	Компонент может копироваться методом PaintTo для рисовки произвольной канве
csNoStdEvents	Игнорируются стандартные события, такие, как нажатие кнопок мыши, клавиш, щелчки. Этот флаг надо устанавливать, если ваш код не должен реагировать на эти события; в результате ваше приложение будет выполняться быстрее
csDisplayDrag-Image	Компонент может отображать изображение из списка изображений, когда мышь перемещается на него. Этот флаг устанавливается, если компонент реализует список изображений для отображения при перемещении на него мыши
csReflector	Компонент реагирует на сообщения Windows, поступающие из диалогов, сообщения о фокусировке, сообщения об изменении размеров. Этот флаг устанавливается, если компонент может использоваться как элемент ActiveX и должен реагировать на эти события
csActionClient	Компонент связан с объектом действия. Этот флаг устанавливается при установке свойства Action и сбрасывается при очистке Action
csMenuEvents	Компонент отвечает на команды главного меню

Свойство **ControlStyle** описывает не свойства отдельных экземпляров класса, а класс в целом. Флаги не могут изменяться для различных экземпляров компонентов и не могут изменяться в процессе выполнения приложения. Изменяемые характеристики отображаются свойством **ControlState**.

Конструктор класса **TControl** инициализирует свойство **ControlStyle** значениями [**csCaptureMouse**, **csClickEvents**, **csSetCaption**, **csDoubleClicks**].

CopyMode

Определяет режим копирования графического изображения на канву

Класс *TCanvas***Определение**

```
__property int CopyMode
```

Описание

Свойство канвы **CopyMode** определяет режим копирования графического изображения на канву методом **CopyRect** или при рисовании объекта **TBitmap**. Используя свойство можно достичь различных эффектов объединения изображений и их комбинирования.

Возможны следующие значения свойства **CopyMode** (используемые константы определены в *Windows.hpp*):

cmBlackness	Заполняет область канвы, в которую производится копирование, черным цветом. Собственное изображение на канве и копируемое изображение игнорируются
cmDstInvert	Инвертирует изображение на канве. Копируемое изображение игнорируется
cmMergeCopy	Комбинирует изображение канвы и копируемое изображение, используя булеву операцию and . То же, что cmSrcAnd
cmMergePaint	Комбинирует изображение канвы и инверсию копируемого изображения, используя булеву операцию or
cmNotSrcCopy	Копирует на канву инверсное изображение. Собственное изображение на канве игнорируется
cmNotSrcErase	Комбинирует изображения канвы и копируемого изображения, используя булеву операцию or , а затем инвертирует результат
cmPatCopy	Копирует шаблон источника на канву. Собственное изображение на канве игнорируется
cmPatInvert	Комбинирует изображение канвы и шаблон источника, используя булеву операцию xor
cmPatPaint	Комбинирует инверсное изображение источника и его шаблон, используя булеву операцию or . Затем этот результат комбинирует с изображением канвы, используя булеву операцию or
cmSrcAnd	Комбинирует изображения канвы и источника, используя булеву операцию and . То же, что cmMergeCopy
cmSrcCopy	Копирует изображение источника на канву. Собственное изображение на канве игнорируется. Этот режим принят по умолчанию
cmSrcErase	Инвертирует изображение на канве и комбинирует результат с изображением источника, используя булеву операцию and
cmSrcInvert	Комбинирует изображения канвы и источника, используя булеву операцию xor . Повторное копирование восстанавливает прежнее изображение на канве
cmSrcPaint	Комбинирует изображения канвы и источника, используя булеву операцию or
cmWhiteness	Заполняет область канвы, в которую производится копирование, белым цветом. Собственное изображение на канве и копируемое изображение игнорируются

Примеры

Операторы

```
Image1->Canvas->CopyMode = cmSrcCopy;
Image1->Canvas->CopyRect (Rect (0,0,100,100), Image2->Canvas,
                           Rect (0,0,100,100));
```

обеспечивают копирование области изображения канвы компонента **Image2** на канву компонента **Image1**. Изображение, которое ранее было на канве компонента **Image1**, в операциях не участвует.

Операторы

```
Image1->Canvas->CopyMode = cmSrcInvert;
Image1->Canvas->CopyRect (Rect (0,0,100,100), Image2->Canvas,
                           Rect (0,0,100,100));
...
Image1->Canvas->CopyRect (Rect (0,0,100,100), Image2->Canvas,
                           Rect (0,0,100,100));
```

обеспечивают копирование части изображения канвы компонента **Image2** на канву компонента **Image1** в режиме **cmSrcInvert**. После выполнения функции **CopyRect** в первый раз изображения в компонентах **Image1** и **Image2** налагаются друг на друга, а в результате выполнения функции **CopyRect** во второй раз исходное изображение на канве компонента **Image1** восстанавливается.

Операторы

```
Image1->Canvas->CopyMode = cmWhiteness;
Image1->Canvas->CopyRect (Rect (0,0,100,100), Image2->Canvas,
                           Rect (0,0,100,100));
```

просто очищают указанную область канвы компонента **Image1**, закрашивая ее белым цветом. При этом изображение в компоненте **Image2** никак не участвует в операциях копирования.

Count

Количество элементов массива указателей, хранящихся в объекте класса **TList** или **TStringList**

Классы *TList*, *TStringList*, *TStrings*

Определение

```
__property int Count
```

Описание

Свойство **Count** показывает число элементов массива указателей, хранящихся в объекте класса **TList** или **TStringList**. При добавлении или удалении элементов методами **Add** и **Delete** значение **Count** увеличивается или уменьшается автоматически. Если намеренно увеличить значение **Count**, то в список добавится соответствующее число нулевых указателей **NULL**. Если намеренно уменьшить значение **Count**, то соответствующее число последних элементов списка будет удалено.

Значение **Count** не всегда равно действительному числу указателей на объекты, хранящихся в списке, поскольку ряд хранящихся в нем указателей могут иметь значение **NULL**. Удалить эти элементы **NULL** можно методом **Pack**.

Свойство **Count** отличается от близкого ему свойства **Capacity**. **Capacity** показывает, сколько указателей может храниться в массиве, т.е. емкость списка, а **Count** показывает, сколько указателей в действительности хранится в массиве. Поэтому значение **Count** всегда не больше значения **Capacity**.

В классе **TStrings** **Count** — это абстрактное свойство, но в его наследниках указывает число строк.

Пример

См. пример в разделе **Capacity**.

Ctl3D

Определяет, будет ли компонент выглядеть объемным или плоским

Класс *TWinControl*

Определение

```
__property bool Ctl3D
```

Описание

Свойство **Ctl3D** определяет внешний вид элемента. Если **Ctl3D** установлено в **true**, элемент выглядит объемным. Если же **Ctl3D** установлено в **false**, то элемент выглядит плоским. По умолчанию **Ctl3D** равно **true**.

Если свойство дочерних компонентов **ParentCtl3D** установлено в **true**, то изменение свойства **Ctl3D** родительского компонента автоматически приводит к изменению свойств **Ctl3D** дочерних компонентов. Когда явным образом идет установка свойства **Ctl3D** компонента, его свойство **ParentCtl3D** автоматически устанавливается в **false**.

Чтобы свойство **Ctl3D** работало для радиокнопок, индикаторов и любых обычных диалогов в системе Windows NT 3.51, в каталоге System32 должна быть установлена библиотека CTL3D32.DLL. Для Windows 95/98 и NT 4.0 эта библиотека не требуется.

Cursor

Определяет изображение курсора мыши, когда он расположен в области компонента

Класс *TControl*

Определение

```
enum TCursor {crMin=0x7fff-1, crMax=0x7fff};
```






```
__property TCursor Cursor
```

Описание

Изменение изображения курсора мыши при перемещении его в области компонента указывает пользователю на действия, которые он при этом может совершить. Имеется еще одно свойство, аналогичное **Cursor** — свойство **DragCursor**. Оно отвечает за изображение курсора при перемещении его в области компонента в процессе перетаскивания.

Значения свойств **Cursor** и **DragCursor** являются индексом списка возможных курсоров, управляемого глобальной переменной **Screen**. Кроме встроенных типов курсоров, обеспечиваемых в **TScreen**, приложение может добавить в список свои заказные изображения (см. раздел 4.5.6 главы 4).

Ниже приводится перечень встроенных в **TScreen** типов курсоров:

Значение	Изображение	Значение	Изображение
crDefault	Курсор по умолчанию. Обычно это crArrow .	crHourGlass	
crNone		crDrag	
crArrow		crNoDrop	

Значение	Изображение	Значение	Изображение
crCross		crHSplit	
crIBeam		crVSplit	
crSize		crMultiDrag	
crSizeNESW		crSQLWait	
crSizeNS		crNo	
crSizeNWSE		crAppStart	
crSizeWE		crHelp	
crUpArrow		crHandPoint	

DesktopFont

Определяет, использует ли компонент для отображения текста изображение шрифта Windows

Класс *TControl*

Определение

```
__property bool DesktopFont
```

Описание

Установка свойства **DesktopFont** в **true** определяет, что компонент должен использовать для свойства **Font** изображение шрифта Windows. Этот шрифт задается свойством **IconFont** глобальной переменной **Screen**.

Если **DesktopFont** установлено в **true**, то свойство **Font** будет изменяться всякий раз при изменении шрифта Windows. Это изменение может осуществляться заданием свойства **IconFont** глобальной переменной **Screen** или изменяться другими программами. Установка свойства **Font** напрямую равным **Screen->IconFont** не разрешается.

DockOrientation

Определяет, как в рамках технологии Drag&Doc данный компонент будет встраиваться в контейнер по отношению к другим встроенным компонентам

Класс *TControl*

Определение

```
enum TDockOrientation { doNoOrient, doHorizontal, doVertical };
__property TDockOrientation DockOrientation
```

Описание

Свойство **DockOrientation** определяет позицию, в которой будет встраиваться компонент по отношению к другим встроенным компонентам: **doNoOrient** — без какой-то заданной ориентации, **doHorizontal** — зоны размещения упорядочиваются сверху вниз, **doVertical** — зоны размещения упорядочиваются слева направо.

DragCursor

См. **Cursor**.

DragKind

Определяет, будет ли объект перетаскиваться по технологии Drag&Drop, или Drag&Doc

Класс *TControl*

Определение

```
enum TDragKind { dkDrag, dkDock };  
__property TDragKind DragKind
```

Описание

Свойство **DragKind** определяет, будет ли компонент перетаскиваться обычным образом (по технологии Drag&Drop) — значение **dkDrag**, или он будет перетаскиваться для встраивания (по технологии Drag& Doc) — значение **dkDock**.

DragMode

Определяет поведение компонента в процессе его перетаскивания

Класс *TControl*

Определение

```
enum TDragMode { dmManual, dmAutomatic };  
__property TDragMode DragMode
```

Описание

Свойство **DragMode** определяет поведение компонента в процессе его перетаскивания. Свойство может принимать значения:

dmAutomatic	От программиста не требуется обработка каких-либо событий. Компонент готов к перетаскиванию. Пользователю достаточно щелкнуть на его изображении и начать перетаскивать
dmManual	Компонент не может быть перетащен, пока приложение не вызовет метод BeginDrag

Примеры использования свойства **DragMode** см. в разделе **OnDragDrop**.

DrawingStyle

Определяет стиль, используемый при рисовании изображения

Класс *TCustomImageList*

Определение

```
enum TDrawingStyle { dsFocus, dsSelected, dsNormal, dsTransparent};  
__property TDrawingStyle DrawingStyle
```

Описание

Свойство **DrawingStyle** определяет стиль изображения. Возможные значения:

dsFocused	Изображение рисуется на 25% смешанное с цветом, указанным свойством BlendColor . Влияет только на список изображений, содержащий маски
dsSelected	Изображение рисуется на 50% смешанное с цветом, указанным свойством BlendColor . Влияет только на список изображений, содержащий маски

dsNormal	Значение по умолчанию. Изображение рисуется с использованием цвета, указанного свойством BkColor . Если BkColor задано равным clNone , изображение рисуется прозрачным с использованием маски
dsTransparent	Изображение рисуется с использованием маски, независимо от установки BkColor

Enabled

Определяет, реагирует ли компонент на события, связанные с мышью, клавиатурой и таймером

Класс *TControl*

Определение

`__property bool Enabled`

Описание

Свойство **Enabled** определяет доступность компонента для пользователя. Чтобы сделать компонент недоступным, надо установить **Enabled** в **false**. Недоступный компонент отображается серым цветом. Он игнорирует события, связанные с мышью, клавиатурой и события таймера **OnTimer**.

Чтобы восстановить доступность компонента, надо установить **Enabled** в **true**. Компонент начинает нормально отображаться на экране и реагировать на действия пользователя.

Font

Определяет атрибуты шрифта

Класс *TControl*

Определение

`__property Graphics::TFont* Font`

Описание

Свойство **Font** является объектом типа **TFont**. Изменение шрифта можно осуществить или созданием нового объекта типа **TFont**, или изменением свойств **Color**, **Height**, **Name**, **Pitch**, **Size**, **Style** существующего объекта. См. подробнее методы свойства и события объекта **Font** и его установки по умолчанию в разделе **TFont**.

Примеры

1. Пусть на форме имеется компонент **Memo1**, в котором расположен некоторый текст, и компонент **FontDialog1** — диалог выбора шрифта. Для того, чтобы пользователь мог выбрать имя и атрибуты шрифта текста, отображаемого в **Memo1**, надо вставить в текст оператор:

```
if (FontDialog1->Execute())
    Memo1->Font->Assign(FontDialog1->Font);
```

Если пользователь сменил атрибуты в диалоговом окне выбора шрифта, то метод **FontDialog1->Execute()** возвращает **true** и атрибуты шрифта компонента **Memo1** устанавливаются равными выбранным пользователем.

2. Аналогичный выбор пользователем шрифта, но уже не для всего текста, а только для выделенного фрагмента или для текущего абзаца при использовании компонента **RichEdit** обеспечивается оператором:

```
if (FontDialog1->Execute())
    RichEdit1->SelAttributes->Assign(FontDialog1->Font);
```

3. Для того, чтобы продемонстрировать доступные в системе шрифты, можно построить форму, содержащую выпадающий список **ComboBox1** и компонент **Memo1** с некоторым текстом. Приведенная ниже программа загружает в список доступные имена шрифтов и отображает первую строку списка (этот код вставлен в обработчик события формы **OnCreate**). В обработчик события **OnClick** компонента **ComboBox1** вставлен оператор, меняющий подсвойство **Name** в свойстве **Font** компонента **Memo1**.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    ComboBox1->Items = Screen->Fonts;
    ComboBox1->ItemIndex = 0;
}

void __fastcall TForm1::ComboBox1Click(TObject *Sender)
{
    Memo1->Font->Name = ComboBox1->Items->Strings[ComboBox1->ItemIndex];
}
```

4. Для того, чтобы посмотреть влияние на отображаемый текст свойства **Charset**, определяющего набор символов шрифта, можно добавить на форму компонент редактирования (например, **CSpinEdit**) и кнопку, в обработчик события **OnClick** которой ввести оператор:

```
Memo1->Font->Charset = CSpinEdit1->Value;
```

5. Чтобы посмотреть влияние на отображаемый текст свойства **Pitch**, можно добавить на форму еще один компонент **ComboBox**, задать в его свойстве **Items** строки «fpDefault», «fpFixed» и «fpVariable», а в событие **OnClick** вставить код:

```
switch (ComboBox2->ItemIndex)
{
    case 0: Memo1->Font->Pitch = fpDefault;
            break;
    case 1: Memo1->Font->Pitch = fpFixed;
            break;
    case 2: Memo1->Font->Pitch = fpVariable;
}
}
```

GroupIndex — свойство разделов меню

Определяет логическую группу, к которой относится раздел главного меню и которая используется при объединении меню нескольких форм

Класс *TMenuItem*

Определение

```
__property Byte GroupIndex
```

Описание

Свойство **GroupIndex** определяет способ объединения меню. В MDI приложениях меню дочерних форм всегда объединяются с меню родительской формы. В приложениях с несколькими формами наличие или отсутствие объединения определяется свойством **AutoMerge** компонента типа **TMainMenu**.

По умолчанию все разделы меню имеют одинаковое значение **GroupIndex**. Если требуется объединение меню, то разделам надо задать не убывающие номера свойств **GroupIndex**. Тогда, если разделы встраиваемого меню имеют те же значе-

ния, что и какие-то разделы меню основной формы, то эти разделы заменяют соответствующие разделы основного меню. В противном случае разделы вспомогательного меню встраиваются между элементами основного меню в соответствии с номерами **GroupIndex**. Если встраиваемый раздел имеет **GroupIndex** меньший, чем любой из разделов основного меню, то разделы встраиваются в начало.

Когда активизирован объект, созданный сервером OLE 2.0, сервер может попытаться объединить свое меню с меню контейнера приложения. Тогда свойство **GroupIndex** используется для замены до трех разделов главного меню. Для замещения приложение сервера использует следующие предопределенные значения **GroupIndex**:

Группа	Индекс	Описание
Edit	1	Разделы меню сервера, связанные с редактированием активного объекта OLE
View	3	Разделы меню сервера, связанные с отображением активного объекта OLE
Help	5	Разделы меню сервера, связанные с доступом ко встроенной справке

Свойство **GroupIndex** может использоваться и еще для одной цели: создания группы разделов, работающих по принципам радиокнопок. Для этого во всех разделах группы надо установить в **true** свойство **RadioItem** и задать всем разделам одинаковое значение **GroupIndex**. Выбор пользователем одного из таких разделов будет снимать выделение с остальных разделов.

Примеры использования свойства **GroupIndex** см. в разделе 3.6.1 главы 3.

Handle

Дескриптор окна

Класс *TWinControl*

Определение

`__property HWND Handle`

Доступ только для чтения

Описание

Свойство **Handle** используется при обращении к функциям API Windows, требующим указания дескриптора окна.

Обращение к свойству **Handle** приводит к созданию дескриптора, если его не было до этого. Поэтому нельзя обращаться к этому свойству при создании компонента или чтении его из потока.

Height — свойство компонента

Определяет высоту компонента или формы в пикселях

Класс *TControl*

Определение

`__property int Height`

Описание

Свойство **Height** определяет вертикальный размер компонента или формы в пикселях. Используется для изменения высоты компонента при изменениях размеров окна приложения. См. разделы **ClientHeight**, **ClientRect**.

Height — свойство шрифта

Определяет высоту шрифта в пикселях

Класс *TFont*

Определение

`__property int Height`

Описание

Свойство **Height** определяет высоту шрифта в пикселях. Если значение **Height** задано отрицательным, то в размер не входит верхний пиксель каждой строки.

Обычно для задания размера используется не **Height**, а другое свойство: **Size** — размер шрифта в кеглях или в пунктах, принятых в Windows.

Значение **Height** связано со свойствами **Size** и **PixelsPerInch** (число пикселей на дюйм — см. **TFont**) соотношением:

`Font->Height = -Font->Size * Font->PixelsPerInch / 72`

Из этого соотношения, в частности, видно, что задание положительного значения **Size** ведет к отрицательному значению **Height**. Можно задавать **Size** отрицательным; тогда **Height** будет положительным.

HelpContext

Определяет индекс, используемый в контекстно-зависимой справке

Класс *TWinControl*

Определение

`__property Classes::THelpContext HelpContext`

Описание

Значение **HelpContext** задается для определения уникального индекса темы контекстно-зависимой справки. Экран с этой темой будет показан пользователю, если он нажмет клавишу F1, когда данный компонент в фокусе.

Если значение **HelpContext** равно 0, то компонент наследует значение **HelpContext** своего родительского элемента (элемента-контейнера, в котором он расположен). Например, если задано значение **HelpContext** для формы, то для всех компонентов, у которых не задано отличного от нуля значения **HelpContext**, будет показана тема справки, указанная для формы. Тип **THelpContext** определен в модуле *Classes.cpp* следующим образом

`typedef int THelpContext;`

Hint

Содержит текст, отображаемый в окне подсказки или в строке состояния

Классы *TControl*, *TApplication*

Определение

`__property AnsiString Hint`

Описание

Свойство **Hint** компонента обеспечивает текст подсказки, появляющийся в ярлычке (всплывающем окне подсказки) или в заданном месте окна, например, в строке состояния.

В общем случае **Hint** состоит из двух частей, разделенных символом вертикальной черты '|'. Первая часть отображается в ярлычке, если пользователь задержит курсор мыши над данным компонентом (это может быть любой компонент, включая разделы меню). Обычно первая часть содержит краткое пояснение компо-

нента. В частности, такой подсказкой как правило снабжаются быстрые кнопки типа **TSpeedButton**. Вторая часть содержит текст, отображаемый в какой-то выделенной для этого части окна, например, в строке состояния. Это обычно развернутое пояснение. Например, свойство **Hint** для быстрой кнопки доступа к разделу меню сохранения файла может иметь вид: «Сохранить/Сохранение текущего документа в файле». Как частный случай, в свойстве **Hint** может быть задана только первая часть подсказки без символа '|'.

Для того, чтобы первая часть подсказки появлялась в ярлычке, когда пользователь задержит курсор мыши над данным компонентом, надо сделать следующее:

1. Указать тексты свойства **Hint** для всех компонентов, для которых вы хотите обеспечить окно подсказки.
2. Установить свойства **ShowHint** (показать подсказку) этих компонентов в **true** или установить в **true** свойство **ParentShowHint** (**ShowHint** родителя) и установить в **true** свойство **ShowHint** контейнера, содержащего данные компоненты, или формы.

Конечно, вы можете устанавливать свойства в **true** или **false** программно, включая и отключая подсказки в различных режимах работы приложения.

При **ShowHint**, равном **true**, ярлычок будет всплывать даже если компонент в данный момент недоступен (**Enabled** = **false**).

Если вы не задали значение свойства компонента **Hint**, но установили в **true** свойство **ShowHint** или установили в **true** свойство **ParentShowHint**, а в родительском компоненте **ShowHint** = **true**, то в ярлычке будет отображаться текст **Hint** из родительского компонента.

Правда, все описанное выше справедливо при значении свойства **ShowHint** приложения (объекта **Application**), равном **true** (это значение задано по умолчанию). Если установить **Application->ShowHint** в **false**, то ярлычки не будут появляться, независимо от значений **ShowHint** в любых компонентах.

Для того, чтобы вторая часть сообщения, записанного в **Hint**, отображалась в строке состояния в моменты, когда курсор мыши проходит над компонентом, надо использовать обработку события **OnHint**. Это событие именно приложения **Application**, а не того компонента, над которым проходит курсор мыши. В C++Builder 5 его можно перехватить с помощью компонента **ApplicationEvents**. Если обработчик этого события определен, то в момент прохождения курсора над компонентом, в котором задано свойство **Hint**, вторая часть сообщения компонента заносится в свойство **Hint** объекта **Application**. Если свойство **Hint** компонента содержит только одну часть, то в свойство **Hint** объекта **Application** заносится эта первая часть. Причем все это делается независимо от состояния свойства компонента **ShowHint**.

Третий способ использования свойства **Hint** компонента заключается в непосредственном отображении текста заключенного в нем сообщения в какой-то метке или панели с помощью функций **GetShortHint** и **GetLongHint**, первая из которых возвращает первую часть сообщения, а вторая — вторую (если второй части нет, то возвращается первая часть).

Примеры использования свойства **Hint** приведены в главе 4 в разделе 4.1.9.

HostDockSite

Определяет контейнер, в который встроен данный компонент

Класс *TControl*

Определение

```
__property TWinControl* HostDockSite
```

Описание

Свойство **HostDockSite** определяет контейнер, в который встроен данный компонент. Если компонент находится в состоянии «плавающего» окна, то **HostDockSite** — временный объект типа **FloatingDockSiteClass** или **NULL**.

Обычно для программного встраивания компонента лучше использовать метод **ManualDock**, а не задание **HostDockSite**. Установка **HostDockSite** приводит к немедленному встраиванию компонента в указанный контейнер, но не устанавливает позицию компонента, его выравнивание и не воспроизводит событий встраивания.

Для встроенных компонентов значение **HostDockSite** совпадает со значением **Parent**. Но если компонент никуда не встроен, то **HostDockSite** = **NULL**, а **Parent** указывает на контейнер, содержащий данный компонент.

ImageIndex

Индекс изображения раздела в списке изображений

Классы *TMenuItem*, *TToolButton*, *TListItem*, *TTreeNode*, *TTabSheet* и др.

Определение

`__property int ImageIndex`

Описание

Свойство **ImageIndex** указывает индекс изображения, появляющегося левее надписи данного раздела меню, на кнопке инструментальной панели, в строке списка, узле дерева и т.п. Индекс относится к списку изображений — свойству **Images** компонента-контейнера (родительского меню типа **TMenu** или **TPopupMenu**, списка, панели и т.п.) и начинается с 0. По умолчанию **ImageIndex** = -1, что означает отсутствие изображения.

В компонентах-меню, если не задан список изображений (свойство **Images** равно **NULL**), то для отображения изображения надо использовать свойство **Bitmap** компонента типа **TMenuItem**. Но **Bitmap** действует только при **Images** равном **NULL** и отрицательном значении **ImageIndex**. В противном случае значение **Bitmap** не принимается во внимание.

Items — свойство класса TList

Список ссылок на элементы массива в объекте класса **TList**

Класс *TList*

Определение

`__property void * Items[int Index]`

Описание

Свойство **Items** дает доступ к указателям, хранящимся в объекте класса **TList**. Параметр **Index** является индексом массива указателей. Индексы изменяются, начиная с 0 (0 — индекс первого указателя).

Свойство **Items** позволяет получить доступ и к объектам, на которые указывают элементы массива. Но при этом надо учитывать, что свойство **Items** имеет тип **pointer**, т.е. является нетипизированным указателем. Такой указатель нельзя разыменовывать непосредственно. Надо использовать явное или неявное приведение типов. Примеры этого даны ниже, а также в разделе **TList**.

При организации циклов с просмотром указателей, содержащихся в **Items**, надо иметь в виду, что часть из них может быть равна **NULL**. Если это может нарушить работу алгоритма, то предварительно надо применить к объекту **TList** метод **Pack**, который сократит размер массива, удалив из него нулевые указатели.

Пример

```
// Создание экземпляра списка
TList *List = new TList;

// Создание объектов - строк и занесение их в список:
List->Add((char *)malloc(10));
List->Add((char *)malloc(10));

// Задание значений строк через свойство Items
List->Items[0] = "aaa";
List->Items[1] = "bbb";

// Отображение значений элементов (результат - "aaa" и "bbb")
for (int i = 0; i < List->Count; i++)
    ShowMessage((char *)List->Items[i]);

// Удаление из памяти первого элемента
free(List->Items[0]);

// Удаление указателя на первый элемент списка
// (второй элемент становится первым)
List->Delete(0);

// Отображение значений элементов (результат - "bbb")
for (int i = 0; i < List->Count; i++)
    ShowMessage((char *)List->Items[i]);
```

Left

Координата левого края компонента в пикселях

Класс *TControl*

Определение

```
__property int Left
```

Описание

Свойство **Left** определяет координату левого края компонента в пикселях. Для компонентов за начало отсчета берется левая граница клиентской области родителя (например, панели, если данный компонент расположен на панели, или формы, если компонент расположен непосредственно на форме). Для формы координата **Left** представляет собой горизонтальную координату экрана, отсчитываемую от его левого края.

Свойство **Left** используется при перемещениях и изменениях размеров компонентов (см. примеры в разделах **BoundsRect** и **Components**).

List

Указатель на массив указателей в объекте класса **TList**

Определение

```
typedef void *TPointerList[134217727];
typedef TPointerList *PPointerList;
__property PPointerList List
```

Доступ только для чтения

Описание

Свойство **List** дает непосредственный доступ к массиву **Items** — массиву указателей объекта типа **TList**. Например, **MyList->Items[0]** и **MyList->List[0]** указывают на один и тот же элемент массива.

Mode — свойство TPen

Определяет режим рисования пером на канве

Класс *TPen*

Определение

```
enum TPenMode {pmBlack, pmWhite, pmNop, pmNot, pmCopy,
               pmNotCopy, pmMergePenNot, pmMaskPenNot,
               pmMergeNotPen, pmMaskNotPen, pmMerge,
               pmNotMerge, pmMask, pmNotMask, pmXor, pmNotXor};
```

`__property TPenMode Mode`

Описание

Свойство пера **Mode** определяет, каким образом взаимодействуют цвета пера и канвы. Выбор значения **Mode** позволяет получать различные эффекты.

Возможные значения **Mode**:

Режим	Цвет пикселя
pmBlack	Всегда черный
pmWhite	Всегда белый
pmNop	Неизменный
pmNot	Инверсный по отношению к цвету фона канвы
pmCopy	Цвет, указанный в свойстве Color пера Pen : это значение принято по умолчанию
pmNotCopy	Инверсия цвета пера
pmMergePenNot	Комбинация цвета пера и инверсного цвета фона канвы
pmMaskPenNot	Комбинация цветов, общих для цвета пера и инверсного цвета фона канвы
pmMergeNotPen	Комбинация цвета фона канвы и инверсного цвета пера
pmMaskNotPen	Комбинация цветов, общих для цвета фона канвы и инверсного цвета пера
pmMerge	Комбинация цвета пера и цвета фона канвы
pmNotMerge	Инверсия режима pmMerge : комбинации цвета пера и цвета фона канвы
pmMask	Комбинация цветов, общих для цвета фона канвы и цвета пера
pmNotMask	Инверсия режима pmMask : комбинации цветов, общих для цвета фона канвы и цвета пера
pmXor	Операция xor : комбинация цветов или пера, или фона канвы, но не обоих
pmNotXor	Инверсия режима pmXor : комбинации цветов или пера, или фона канвы, но не обоих

Name

Имя компонента, по которому на него ссылаются другие компоненты

Класс *TComponent*

Определение

__property AnsiString Name

Описание

Свойство **Name** определяет имя компонента, которое используется в дальнейшем в программе. Задается только в процессе проектирования и не должно изменяться во время выполнения. Иначе операторы программы могут оказаться неправильными.

По умолчанию C++Builder сама присваивает имена компонентам, размещаемым на форме. Эти имена по умолчанию надо изменять в процессе проектирования на осмысленные. Иначе при сколько-нибудь сложной форме вы сами через некоторое время не сможете понять, что такое **Panel7** или **Button13**.

Пример использования свойства **Name** см. в разделе **Components**.

Parent

Определяет родительский компонент, в площади которого располагается данный компонент

Класс *TControl*

Определение

__property TWinControl* Parent

Описание

Свойство **Parent** определяет родительский компонент, т.е. компонент-контейнер, содержащий данный компонент. Контейнерами являются оконные компоненты, такие, как формы, панели и некоторые другие. Расположенные на них дочерние компоненты могут наследовать часть свойств содержащего их контейнера, например, шрифт, цвет, отображение ярлычков подсказки, трехмерность. Для этого должны быть установлены в **true** свойства дочерних компонентов **ParentFont**, **ParentColor**, **ParentShowHint**, **ParentCtl3D**.

Надо различать два похожих свойства: **Parent** — родительский компонент, и **Owner** — владелец компонента. Родительский компонент — это тот, на котором располагается данный компонент. А владелец — это компонент, который передается в качестве параметра в конструктор данного компонента и который владеет им. Форма является владельцем всех расположенных на ней компонентов. В свою очередь объект приложения **Application** является владельцем всех форм.

Изменение во время выполнения свойства **Parent** заставляет компонент перемещаться на экране в клиентскую область нового родителя.

Если в приложении программно создается новый визуальный компонент, он не будет виден, пока в нем не задано значение **Parent**.

Пример использования свойства **Parent** приведен в разделе **Visible**.

ParentColor

См. раздел **TControl**.

ParentCtl3D

Определяет наследование свойства объемного изображения от родительского компонента

Класс *TWinControl*

Определение

__property bool ParentCtl3D

Описание

Если свойство **ParentCtl3D** имеет значение **true**, то компонент наследует свойство **Ctl3D**, управляющее объемным или плоским изображением, от своего родительского элемента. Это способствует единообразию изображений. Если все компоненты на форме имеют значение **ParentCtl3D**, равное **true**, то их вид определяется значением **Ctl3D** формы.

Непосредственное задание свойства **Ctl3D** какому-либо компоненту автоматически приводит к сбросу на **false** его свойства **ParentCtl3D**.

ParentFont

Включает и выключает использование шрифта родительского компонента

Класс *TControl*

Определение

`__property bool ParentFont`

Описание

Свойство **ParentFont** определяет, будет ли для данного компонента использоваться шрифт (свойство **Font**) родительского компонента-контейнера. Если установить во всех компонентах, размещенных на панели, **ParentFont** в **true**, то во всех компонентах будет использоваться одинаковый шрифт с одинаковым размером, цветом, стилем. Если все компоненты на форме имеют **ParentFont**, равным **true**, то во всех них атрибуты шрифта определяются свойством **Font** формы. Тогда, например, увеличение размера шрифта у формы приведет к согласованному изменению размеров шрифта всех размещенных на ней компонентов.

Если у формы свойство **ParentFont** тоже установлено в **true**, то шрифт определяется свойством **Font** объекта приложения **Application**.

При изменении свойства **Font** в каком-то компоненте, его свойство **ParentFont** автоматически сбрасывается в **false**.

ParentShowHint

Включает и выключает использование родительского свойства **ShowHint**

Класс *TControl*

Определение

`__property bool ParentShowHint`

Описание

Свойство **ParentShowHint** используется, чтобы иметь возможность одновременно всем компонентам некоторого контейнера или формы разрешать или запрещать отображение ярлычков (всплывающих окон подсказки) при задержке на них курсора мыши. Отображаемый в окнах текст определяется свойствами **Hint** компонентов.

Если свойство **ParentShowHint** установлено в **true**, то разрешение или запрет отображения ярлычков определяется свойством **ShowHint** родительского компонента. Если же свойство **ParentShowHint** установлено в **false**, то отображение окон подсказки определяется свойством **ShowHint** самого компонента.

При задании в компоненте значения **ShowHint**, равного **true**, его свойство **ParentShowHint** автоматически сбрасывается в **false**.

Управлять отображением окон подсказок всего приложения в целом можно также свойством **ShowHint** объекта **Application**.

Подробные пояснения см. в разделе **Hint**.

Pen

Определяет атрибуты пера, используемого для рисования линий и фигур

Класс *TCanvas*

Определение

`__property TPen* Pen`

Описание

Свойство канвы **Pen** определяет атрибуты пера, используемого для рисования линий и фигур. Это свойство является объектом типа **TPen** (см. описание этого типа для получения дополнительной информации). Атрибуты объекта типа **TPen** определяют цвет, ширину, стиль линий и режим рисования пера.

Присваивание свойства **Pen** может производиться методом **Assign**.

PenPos

Определяет положение пера на канве

Класс *TCanvas*

Определение

`__property Windows::TPoint PenPos`

Описание

Свойство канвы **PenPos** определяет переменной типа **TPoint** положение пера на канве. Координаты пера, определенные этим свойством, задают начальную точку рисования линии методом **LineTo**.

Свойство **PenPos** изменяется методом **MoveTo** и некоторыми методами рисования (например, методом **LineTo**). Непосредственная установка **PenPos** эквивалентна применению метода **MoveTo**.

Pitch

Определяет способ установки ширины символов шрифта

Класс *TFont*

Определение

`enum TFontPitch { fpDefault, fpVariable, fpFixed };
__property TFontPitch Pitch`

Описание

Каждый вид шрифта имеет соответствующий способ определения ширины символов. Есть шрифты с одинаковой шириной всех символов. Есть шрифты, в которых разные символы имеют разную ширину. Шрифты с постоянной шириной используются для отображения исходных кодов программ, поскольку в них удобно делать фиксированные отступы. Но шрифты с различной шириной символов выглядят естественнее и более компактны.

Возможные значения свойства **Pitch**:

fpDefault	Ширина устанавливается равной по умолчанию, т.е. описанной в шрифте заданного вида Name
fpFixed	Установка одинаковой ширины всех символов
fpVariable	Установка различной ширины символов

Установка значений **fpVariable** или **fpFixed** заставляет Windows искать наилучший способ удовлетворить всем заданным характеристикам шрифта. Иногда

это может привести к замене шрифта на шрифт другого, близкого вида. Но иногда может вообще не повлиять на шрифт. Все зависит от конкретного вида шрифта и даже от версии этого шрифта.

См. пример 5 в разделе **Font**.

Pixels

Определяет цвета пикселей канвы в пределах текущей области **ClipRect**

Класс *TCanvas*

Определение

```
enum TColor {clMin=-0x7fffffff-1, clMax=0x7fffffff};  
__property TColor Pixels[int X][int Y]
```

Описание

Свойство канвы **Pixels** определяет цвет пикселя канвы с координатами **X** и **Y** в пределах текущей области **ClipRect**. Если заданы координаты пикселя вне области **ClipRect**, то при чтении свойства **Pixels** возвращается значение -1.

Задание значений пикселей позволяет рисовать по пикселям графики и линии. Определение цвета пикселя используется обычно в методе **FillRect**. Подробнее об использовании пикселей см. в главе 5 в разделе 5.1.3.

Не все устройства поддерживают свойство **Pixels**. Чтение **Pixels** для таких устройств возвращает -1. Установка **Pixels** для подобных устройств не дает никаких результатов.

PopupMenu

Определяет всплывающее меню, связанное с данным компонентом

Класс *TControl*

Определение

```
__property Menus::TPopupMenu* PopupMenu
```

Описание

Задание свойства **PopupMenu** обеспечивает появление всплывающего меню, если в то время, когда выбран данный компонент, пользователь щелкнул правой кнопкой мыши. Обычно в этом всплывающем меню задаются основные команды, относящиеся к данному компоненту. Объект меню имеет тип **TPopupMenu**. Если в этом объекте свойство **AutoPopupMenu** установлено в **true**, то меню будет появляться автоматически. В противном случае надо отображать меню с помощью метода **Popup** класса **TPopupMenu**.

Shortcut

Определяет комбинацию «горячих» клавиш, обеспечивающих быстрый доступ к разделу меню

Класс *TMenuItem*

Определение

```
typedef Word TShortcut;  
__property TShortcut Shortcut
```

Описание

Задание свойства **Shortcut** позволяет пользователю не выбирать данный раздел из меню, а просто нажать заданную комбинацию «горячих» клавиш. Эта комбинация при установке **Shortcut** автоматически появляется в надписи раздела.

При задании свойства **Shortcut** во время проектирования Инспектор Объектов предлагает длинный список возможных комбинаций клавиш. При задании

значения **ShortCut** во время выполнения можно использовать функции **ShortCut**, **TextToShortCut**, **ShortCutToText** (см. раздел 15.7.4 главы 15)

ShowHint

Включает или отключает показ ярлычка (всплывающего окна подсказки) при задержке курсора мыши над компонентом

Классы *TControl, Application*

Определение

`__property bool ShowHint`

Описание

Ярлычок (всплывающее окно подсказки) при задержке курсора мыши над компонентом появляется, если свойство компонента **ShowHint** (показать подсказку) установлено в **true** и задан текст подсказки в свойстве **Hint**. Правда, и при **ShowHint = false** всплывающее окно подсказки может появляться, если установлено в **true** свойство **ParentShowHint** (взять **ShowHint** родительского компонента), а в родительском компоненте **ShowHint = true**.

Изменение **ShowHint** автоматически ставит **ParentShowHint** в **false**.

Свойство **ShowHint** приложения **Application** (по умолчанию равно **true**) определяет, могут ли появляться окна подсказки в каких-то компонентах. Если установить **Application->ShowHint** в **false**, то окна подсказки не будут появляться независимо от значений **ShowHint** в любых компонентах.

См. подробные пояснения в **Hint**.

Showing

Определяет, виден ли компонент в данный момент

Класс *TWinControl*

Доступ *только для чтения*

Определение

`__property bool Showing`

Описание

Свойство **Showing** показывает, может ли пользователь в данный момент видеть компонент. Правда, при этом не учитывается, что другие компоненты могут загородить данный. Поэтому значение **Showing**, равное **true**, еще не гарантирует, что пользователь действительно видит компонент. Если компонент накрыт другим видимым компонентом, то пользователь его все-таки не увидит.

Если свойство **Visible** компонента и всех его родителей (компонентов, содержащих его) равно **true**, то и **Showing** равно **true**. Если же свойство **Visible** компонента или какого-то из его родителей равно **false**, то **Showing** равно **false**.

Size

Размер шрифта в кеглях (пунктах)

Класс *TFont*

Определение

`__property int Size`

Описание

Свойство **Size** определяет размер шрифта в кеглях (пунктах, принятых в Windows). Если значение **Size** задано отрицательным, то в размер входит верхний пиксель каждой строки. Если значение **Size** задано положительным, то этот пиксель не учитывается.

Для задания размера шрифта может использоваться другое свойство: **Height** — размер шрифта в пикселях. Значение **Size** связано со свойствами **Height** и **PixelsPerInch** (число пикселей на дюйм — см. **TFont**) соотношением:

$$\text{Font} \rightarrow \text{Size} = -\text{Font} \rightarrow \text{Height} * 72 / \text{Font} \rightarrow \text{PixelsPerInch}$$

Из соотношения, в частности, видно, что задание положительного значения **Size** ведет к отрицательному значению **Height**. Можно задавать **Size** отрицательным; тогда **Height** будет положительным.

Style — свойство TPen

Определяет стиль рисования линий пером

Класс *TPen*

Определение

```
enum TPenStyle {psSolid, psDash, psDot, psDashDot,
                psDashDotDot, psClear, psInsideFrame};
__property TPenStyle Style
```

Описание

Свойство пера **Style** определяет вид линии. Это свойство может принимать следующие значения:

psSolid	Сплошная линия
psDash	Штриховая линия
psDot	Пунктирная линия
psDashDot	Штрих-пунктирная линия
psDashDotDot	Линия, чередующая штрих и два пунктира
psClear	Отсутствие линии
psInsideFrame	Сплошная линия, но при Width > 1 допускающая цвета, отличные от палитры Windows

Примеры линий всех стилей приведены в главе 5 в разделе 5.1.3.3 на рис. 5.8.

Все стили со штрихами и пунктирами доступны только при **Width** = 1. В противном случае линии этих стилей рисуются как сплошные.

Стиль **psInsideFrame** — единственный, который допускает произвольные цвета. Цвет линии при остальных стилях округляется до ближайшего из палитры Windows.

Style — свойство TBrush

Определяет шаблон заполнения кисти **Brush**

Класс *TBrush*


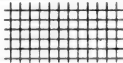
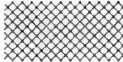

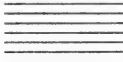


Определение

```
enum TBrushStyle {bsSolid, bsClear, bsHorizontal, bsVertical,
                  bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross};
__property TBrushStyle Style
```

Описание

Свойство кисти **Style** определяет шаблон, которым рисует кисть **Brush**, если для нее не задано значение свойства **Bitmap**.

Возможные значения **Style**:

Значение	Шаблон	Значение	Шаблон
bsSolid		bsCross	
bsClear		bsDiagCross	
bsBDiagonal		bsHorizontal	
bsFDiagonal		bsVertical	

Пример

Код

```
Image1->Canvas->Brush->Color = clRed;  
Image1->Canvas->Brush->Style = bsDiagCross;  
Image1->Canvas->Ellipse(0, 0, Image1->Width, Image1->Height);
```

строит на канве компонента **Image1** эллипс, заполненный красной штриховкой крест на крест.

Style — свойство TFont

Стиль шрифта

Класс *TFont*

Определение

```
enum TFontStyle { fsBold, fsItalic, fsUnderline, fsStrikeOut };  
typedef Set<TFontStyle, fsBold, fsStrikeOut> TFontStyles;  
  
__property TFontStyles Style
```

Описание

Свойство **Style** задает стиль: характер начертания символов заданного шрифта. Свойство представляет собой множество типа **Set**, или пустое, или содержащее одно и более следующих значений:

Значение	Описание
fsBold	Полужирный
fsItalic	Курсив
fsUnderline	Подчеркнутый
fsStrikeout	Перечеркнутый горизонтальной прямой

TabOrder

Определяет позицию компонента в последовательности табуляции

Класс *TWinControl*

Определение

```
typedef short TTabOrder;  
__property TTabOrder TabOrder
```

Описание

Под последовательностью табуляции понимается последовательность, в которой переключается фокус между компонентами окна, когда пользователь последовательно нажимает клавишу табуляции `Tab`. Значение **TabOrder**, равное нулю, означает, что при первом появлении формы на экране в фокусе будет этот компонент.

Последовательность табуляции в каждом приложении надо продумывать, чтобы пользователю было легче работать и переходить от одного окна редактирования к другому, от одной кнопки к другой.

Первоначальная последовательность табуляции определяется просто той последовательностью, в которой размещались управляющие элементы на экране. Первому элементу присваивается значение **TabOrder**, равное 0, второму 1 и т.д.

Если задать какому-то управляющему элементу значение **TabOrder**, равное -1, то этот элемент выпадает из последовательности табуляции и с помощью клавиши `Tab` ему будет невозможно передать фокус.

Каждый управляющий элемент имеет уникальный номер **TabOrder** внутри своего родительского компонента. Поэтому изменение значения **TabOrder** какого-то элемента на уже существующее у другого элемента значение приведет к тому, что значения **TabOrder** всех последующих элементов автоматически изменятся, чтобы не допустить дублирования. Если задать элементу значение **TabOrder**, большее, чем число элементов в родительском компоненте, он просто станет последним в последовательности табуляции.

В среде проектирования C++Builder имеется специальная команда `Edit | Tab Order`, позволяющая в режиме диалога задать последовательность табуляции всех элементов.

Значение свойства **TabOrder** играет роль только тогда, когда свойство компонента **TabStop** установлено в **True** и если компонент имеет родителя. Например, для формы свойство **TabOrder** имеет смысл только в случае, если для формы задан родитель в виде другой формы.

TabStop

Определяет возможность передать фокус на элемент нажатием клавиши табуляции

Класс *TWinControl*

Определение

`__property bool TabStop`

Описание

Свойство **TabStop** разрешает или запрещает останавливать фокус на данном элементе при нажатии пользователем клавиши `Tab`. Если значение **TabStop** равно **true**, то клавиша `Tab` будет передавать фокус на этот элемент в последовательности табуляции, определяемой значениями свойства элементов **TabOrder**. Если значение **TabStop** равно **false**, то элемент недостижим в последовательности табуляции независимо от значения **TabOrder**.

Для формы свойство **TabStop** имеет смысл только в случае, если форма имеет родителя в виде другой формы.

Tag

Свойство, используемое пользователем по своему усмотрению

Класс *TComponent*

Определение

`__property int Tag`

Описание

Свойство **Tag** системой не используется. Пользователь может определить и использовать его по своему усмотрению, помещая в него необходимую информацию. Например, можно в процессе проектирования или программно в процессе выполнения задать некоторое значение **Tag** группе компонентов и затем оперировать с этой группой.

Пример использования свойства **Tag** см. в разделе **Components**.

Text

Текстовая строка, связанная с управляющим элементом

Класс *TControl*

Определение

`__property AnsiString Text`

Описание

Свойство **Text** позволяет прочесть или задать строку, связанную с данным управляющим элементом. По умолчанию значение **Text** равно имени компонента — его свойству **Name**. Применяется в основном в компонентах редактирования и в списках.

TextFlags

Определяет способ вывода текста на канву

Класс *TCanvas*

Определение

`__property int TextFlags`

Описание

Свойство канвы **TextFlags** определяет особенности вывода текста на канву методами **TextOut** и **TextRect**. Свойство **TextFlags** может формироваться как целая комбинация любых следующих констант:

Константа	Пояснение
ETO_CLIPPED	Выводится только текст, помещающийся в указанной прямоугольной области. В методе TextRect этот флаг устанавливается автоматически. На метод TextOut этот флаг не влияет.
ETO_OPAQUE	Текст выводится с непрозрачным цветом фона.
ETO_RTLREADING	Строка текста выводится справа налево. Доступно только с версией Windows Mideast (для стран востока).
ETO_GLYPH_INDEX	Текст является массивом кодов символов, который непосредственно передается GDI Windows. Применимо только для шрифтов TrueType, но этот флаг можно применять и для других шрифтов, чтобы указать, что GDI должен обрабатывать текст напрямую, без языковой обработки. Подробнее см. в документации по GDI Windows.
ETO_IGNORELANGUAGE	Недокументированный пока флаг Microsoft.
ETO_NUMERICSLOCAL	Недокументированный пока флаг Microsoft.
ETO_NUMERICSLATIN	Недокументированный пока флаг Microsoft.

Top

Координата верхнего края компонента в пикселях

Класс *TControl*

Определение

`__property int Top`

Описание

Свойство **Top** определяет координату верхнего края компонента в пикселях. Для компонентов за начало отсчета берется верхняя граница клиентской области родителя (например, панели, если данный компонент расположен на панели, или формы, если компонент расположен непосредственно на форме). Отсчет координаты ведется сверху вниз. Для формы координата **Top** представляет собой вертикальную координату экрана, отсчитываемую от его верхнего края.

Свойство **Top** используется при перемещениях и изменениях размеров компонентов (см. примеры в разделах **BoundsRect** и **Components**).

TransparentColor

Определяет, какой цвет в битовой матрице будет прозрачным при ее рисовании

Класс *TBitmap*

Определение

`__property TColor TransparentColor`

Описание

Значение свойства канвы **TransparentColor** имеет тип **TColor** и зависит от установки свойства **TransparentMode**. Если **TransparentMode** установлено в **tmAuto**, то **TransparentColor** возвращает цвет пикселя левого нижнего угла изображения. Если вы задаете значение свойства **TransparentColor**, то **TransparentMode** автоматически устанавливается в **tmFixed**. При этом новый цвет сохраняется вместе с объектом битовой матрицы и может быть использован позднее. Если вы хотите отменить заданное значение **TransparentColor**, установите **TransparentMode** в **tmAuto**, и тогда **TransparentColor** опять будет указывать на цвет левого нижнего пикселя.

TransparentMode

См. **TransparentColor**.

Visible

Определяет, видим или невидим компонент

Класс *TControl*

Определение

`__property bool Visible`

Описание

Свойство **Visible** определяет видимость компонента во время выполнения. Если **Visible** делается равным **true**, то компонент становится видимым; если **Visible** делается равным **false**, то компонент становится невидимым, исчезает для пользователя. Если устанавливается в **false** свойство **Visible** компонента-контейнера, то становятся невидимыми и все расположенные на нем дочерние компоненты, независимо от значения их свойств **Visible**. Если свойство **Visible** ранее невидимого компонента-контейнера устанавливается в **true**, то становятся видимыми и все его дочерние компоненты, у которых **Visible = true**.

Свойство **Visible** позволяет проектировать на одном и том же месте формы несколько панелей, соответствующих различным режимам работы приложения, и в нужные моменты делать одну из них видимой, а остальные невидимыми, как в приведенном ниже примере.

Свойство **Visible** может также активно использоваться для разделов меню. Очевидно, что обычно не все разделы меню имеют смысл при любых режимах работы приложения. Ненужные разделы можно делать недоступными задавая значения **false** их свойствам **Enabled**. В этом случае они будут видны серыми и недоступными, но размер меню не изменится. А если их делать невидимыми, то они видны не будут, оставшиеся разделы меню сомкнутся и все будет выглядеть более компактно.

Прямое задание значений **true** и **false** свойству **Visible** можно заменить вызовами методов **Show** и **Hide**. Первый из них делает компонент видимым и устанавливает **Visible** в **true**. А второй делает компонент невидимым и устанавливает **Visible** в **false**.

Пример

Пусть в приложении в одном и том же месте формы друг на друге расположены две панели: **Panel1** и на ней **Panel2**, содержащие какие-то управляющие компоненты для разных режимов работы. **Panel2** расположена на **Panel1**, которая является, таким образом, ее родителем. В обработчик события формы **OnCreate** можно вставить операторы:

```
Panel2->Visible = false;
Panel1->Visible = true;
Panel2->Parent = Form1;
Panel2->BoundsRect = Panel1->BoundsRect;
```

Первый два из них делают панель **Panel2** невидимой, а **Panel1** — видимой. Впрочем можно было бы обойтись и без этих операторов, если задать в процессе проектирования значения **Visible**, равными **true** для **Panel1** и **false** для **Panel2**. Третий оператор делает родительским компонентом панели **Panel2** форму **Form1**. А четвертый оператор задает панели **Panel2** то же местоположение и размеры, которые имеет панель **Panel1**. Последнее необходимо, поскольку при проектировании ее координаты соответствовали координатному пространству контейнера — клиентской области панели **Panel1**. А теперь ее родитель сменился на форму, и надо ее расположить в том же месте формы, в котором расположена **Panel1**.

Приведенный код можно сократить, если в процессе проектирования размещать панель **Panel2** не на панели **Panel1**, а в каком-то другом месте непосредственно на форме, и задать значения **Visible**, равными **true** для **Panel1** и **false** для **Panel2**. Тогда в обработчике события формы **OnCreate** достаточно одного оператора:

```
Panel2->BoundsRect = Panel1->BoundsRect;
```

изменяющего положение **Panel2**.

Аналогичный оператор может быть также реализован методом **SetBounds**:

```
Panel2->SetBounds(Panel1->Left, Panel1->Top,
                  Panel1->Width, Panel1->Height);
```

В результате работы одного из приведенных операторов в момент создания формы на ней будет видна панель **Panel1**. В момент, когда ее надо заменить на **Panel2**, можно выполнить операторы:

```
Panel1->Visible = false;
Panel2->Visible = true;
```

делающие невидимой первую и видимой вторую панель. Когда надо вернуть на экран изображение **Panel1**, можно выполнить операторы:

```
Panel2->Visible = false;
Panel1->Visible = true;
```

Другой способ решения той же задачи приведен в разделе **BringToFront**.

Width

Определяет горизонтальный размер компонента или формы в пикселях

Класс *TControl*

Определение

`__property int Width`

Описание

Свойство **Width** определяет горизонтальный размер компонента или формы в пикселях. Используется для изменения ширины компонента при изменениях размеров окна приложения. На компоненты — таблицы во время выполнения изменение **Width** не действует. См. разделы **ClientWidth**, **ClientRect**.

WindowText

Содержит строку текста, связанного с компонентом

Класс *TControl*

Определение

`__property char* WindowText`

Описание

Свойство **WindowText** используется, чтобы связать с компонентом некоторую строку текста, которая может заменяться во время выполнения. По умолчанию **WindowText** — та же самая строка, которая записана в свойстве **Text**. Однако, в классах, производных от **TControl**, это может быть изменено. Для окон редактирования эта строка соответствует отображаемому в компоненте тексту. Для выпадающих списков это текст в окошке редактирования. Для кнопок это имя кнопки. Для остальных компонентов это строка заголовка окна.

16.2 Методы

Add

Функция добавляет новый элемент в список

Классы *TList*, *TStringList*, *TStrings*

Прототипы

Для **TList**:

`int __fastcall Add(void * Item);`

для **TStrings** и **TStringList**:

`virtual int __fastcall Add(const System::AnsiString S);` /

Описание

Функция **Add** добавляет новый элемент в список. Если список не сортированный, то элемент добавляется в конец списка. Если же список сортированный, то новый элемент добавляется в позицию, которая определяется сортировкой. Функция возвращает индекс добавленного элемента (индекс первого элемента — 0). Увеличивает значение свойства **Count** на 1. Если значение **Count** равно значению **Capacity** (емкости массива), то увеличивается значение **Capacity** (с запасом) и перераспределяется память под новые элементы.

Для сортированного списка **TStringList** при выполнении **Add** генерируется исключение **EListError**, если строка **S** уже имеется в списке и свойство **Duplicates** установлено в **dupError**.

Примеры

1. `TList *List = new TList;`

```
// Создание объекта - строки и занесение ее в список:
List->Add((char *)malloc(10));
```

2. `TStringList *TL = new TStringList;` // Создание списка
`TL->Sorted = true;` // Список сортированный
`TL->Duplicates = dupError;` // Запрет дубликатов
`TL->Add("Петров");`
`int i = TL->Add("Иванов");` // Т.к. список сортированный, i=0
`TL->Add("Иванов");` // Генерация исключения из-за дубликата

В приведенном коде оператор

```
int i = TL->Add("Иванов");
```

присваивает переменной `i` значение 0, поскольку список сортированный и фамилия «Иванов» должна размещаться первой, раньше фамилии «Петров». Добавление в сортированный список дубликата вызывает генерацию исключения и сообщение: «String list does not allow duplicates».

Assign — метод графических объектов

Копирует изображение одного графического объекта в другой

Классы *TBitmap*, *TIcon*, *TMetaFile*, *TPicture*

Прототип

```
virtual void __fastcall Assign(Classes::TPersistent * Source);
```

Описание

Метод **Assign** копирует изображение, содержащееся в объекте **Source**, в данный объект. Типы объектов источника и приемника должны быть одинаковыми. Исключение составляет свойство **Graphic** объекта **TPicture**. **Graphic** может участвовать в обменах изображениями с объектами типов **TBitmap**, **TIcon**, **TMetaFile**.

Объектом копирования для классов **TBitmap**, **TIcon**, **TMetaFile** может быть также буфер обмена — объект **Clipboard**. При этом надо не забыть включить в приложение директиву

```
#include <vcl\Clipbrd.hpp>.
```

Свойство **Graphic** объекта **TPicture** может участвовать только в копировании в буфер обмена, но не в копировании из буфера.

Примеры

1. Два приведенных ниже оператора делают одно и то же: копируют изображение из компонента **Image2** в компонент **Image1**. Но второй выполняется успешно только в том случае, если тип графического объекта в **Image2** — **TBitmap**.

```
Image1->Picture->Bitmap->Assign(Image2->Picture->Bitmap);
Image1->Picture->Bitmap->Assign(Image2->Picture->Graphic);
```

2. Каждый из приведенных ниже операторов копирует изображение из компонента **Image2** в буфер обмена **Clipboard**.

```
Clipboard()->Assign(Image2->Picture->Bitmap);
Clipboard()->Assign(Image2->Picture->Graphic);
```

3. Приведенный ниже оператор читает изображение из буфера обмена Clipboard в компонент **Image1**. Если в Clipboard хранится не битовая матрица, будет генерироваться исключение.

```
Image1->Picture->Bitmap->Assign(Clipboard());
```

Assign — метод копирования объектов

Копирует один объект в другой, создавая копию всех данных объекта

Классы *TBlobField*, *TBrush*, *TCheckConstraint*, *TClipboard*, *TCollection*, *TColumn*, *TColumnTitle*, *TControlScrollBar*, *TCoolBand*, *TCustomImageList*, *TDateTimeColors*, *TDimensionItems*, *TField*, *TFieldDefs*, *TFont*, *THeaderSection*, *TIndexDefs*, *TJPEGImage*, *TListColumn*, *TListItems*, *TOleGraphic*, *TParaAttributes*, *TParam*, *TParams*, *TPen*, *TPersistent*, *TSmallIntArray*, *TStatusPanel*, *TstringGridStrings*, *TStrings*, *TTextAttributes*, *TTreeNode*, *TTreeNodees*

Определение

```
<объект-назначение>->Assign(<объект-источник>);
```

Описание

Метод **Assign** копирует данные одного объекта в другой. Объявлен в классе **TPersistent** и перегружен в классах, производных от него. Некоторое число классов C++Builder поддерживает присваивание объектов разных типов. Для большинства же классов, производных от **TPersistent**, применение **Assign** к несовпадающим типам объектов источника и назначения ведет к генерации исключения **EConvertError**.

Метод **Assign** отличается по результатам от операции присваивания

```
<объект-назначение> = <объект-источник>;
```

При присваивании указатель на <объект-назначение> начинает указывать на <объект-источник>. А метод **Assign** создает новую копию объекта. После применения **Assign** имеется два объекта с одинаковыми данными.

Если объекты разного типа, то при вызове **D->Assign(S)** тип **D** должен «знать», как скопировать в него тип **S** (тип **S** может ничего не знать о преобразовании типов). Если метод **Assign** не может осуществить преобразование типов, то он вызывает защищенный метод **AssignTo**, объявленный в классе **TPersistent** и перегруженный в классах, производных от него. Вызов имеет вид **S->AssignTo(D)**. Если и метод **AssignTo** не может осуществить преобразование или если он не перегружен, то вызывается **AssignTo** класса **TPersistent** и генерируется исключение.

Примеры

1. На форме имеется компонент **FontDialog1**, позволяющий пользователю выбрать вид шрифта для изображения надписей на форме. Тогда обработчик соответствующего события в разделе меню может иметь вид:

```
if (FontDialog1->Execute())
    Font->Assign(FontDialog1->Font);
```

2. В программе объявлено и заполнено два списка **SL1** и **SL2** типа **TStringList**. На форме имеется компонент **ComboBox1**, в котором надо отображать один из списков **SL1** или **SL2** в зависимости от того, какая кнопка с флажком нажата в группе радиокнопок **RadioGroup1**. Тогда в событие **OnClick** этой группы радиокнопок надо вставить обработчик, показанный ниже. Последний оператор **ComboBox1->ItemIndex = 0** необходим, чтобы изменение списка сразу отобразилось на экране.

```
TStringList *SL1 = new TStringList;
TStringList *SL2 = new TStringList;
...
void __fastcall TForm1::RadioGroup1Click(TObject *Sender)
{
    if(RadioGroup1->ItemIndex == 0)
        ComboBox1->Items->Assign(SL1);
    else ComboBox1->Items->Assign(SL2);
    ComboBox1->ItemIndex = 0;
}
```

3. Метод **Assign** позволяет проводить обмен данными между совершенно разнородными компонентами, например, компонентом буфера обмена **TClipboard** и графическим объектом **TBitmap**. Записать изображение в буфер можно оператором

```
Clipboard()->Assign(Bitmap);
```

а прочитать из него изображение можно оператором

```
Bitmap->Assign(Clipboard());
```

BeginDrag

Начало процесса перетаскивания компонента

Класс *TControl*

Прототип

```
void __fastcall BeginDrag(bool Immediate, int Threshold);
```

Описание

Метод **BeginDrag** вызывается, когда начинается процесс перетаскивания компонента. Его необходимо вызывать только если значение свойства **DragMode** компонента равно **dmManual**. В противном случае процесс перетаскивания производится автоматически.

Вызов метода **BeginDrag** целесообразно вставлять в обработчик события **OnMouseDown**. Параметр **Immediate** (немедленно) определяет, сразу ли после нажатия кнопки мыши ее указатель изменит вид на тот, который задан свойством **DragCursor**, и сразу ли начнется процесс перетаскивания. Если параметр **Immediate** задан равным **false**, то указатель мыши не изменяет свой вид и процесс перетаскивания не начинается, пока пользователь не сместит указатель на число пикселей, заданное параметром **Threshold**. Это позволяет компоненту воспринимать щелчок мыши, не начиная операцию перетаскивания.

Если значение **Threshold** задано отрицательным (это принято по умолчанию), то функция **BeginDrag** использует значение свойства **DragThreshold** глобальной переменной **Mouse**.

Пример применения метода см. в разделе **OnDragDrop**.

BringToFront

Перенос компонента на верх Z-последовательности

Класс *TControl*

Прототип

```
void __fastcall BringToFront(void);
```

Описание

Метод **BringToFront** позволяет изменять последовательность перекрытия компонентов на форме и тем самым управлять видимостью компонентов.

Перекрывающиеся компоненты на форме размещаются поверх друг друга в так называемой Z-последовательности, соответствующей порядку размещения компонентов в процессе проектирования. Например, если вы поместили в одно и то же место формы две кнопки одинаковых размеров, то видна будет только вторая из размещенных кнопок, поскольку она расположена в Z-последовательности выше. Применение во время выполнения приложения метода **BringToFront** к нижней кнопке переместит ее наверх в Z-последовательности и она станет видна пользователю.

Это справедливо по отношению к неоконным объектам, таким, как кнопки, метки, изображения и т.д., а также и к оконным компонентам, таким, как **Мемо**, **ComboBox** и др. Но все неоконные компоненты всегда расположены в Z-последовательности ниже оконных и метод **BringToFront** не может изменить это правило. Например, попытка перенести наверх методом **BringToFront** метку, размещенную под оконным компонентом, ни к чему не приведет.

Примеры

1. Пусть вы хотите, чтобы в каком-то месте формы размещалась кнопка, которая в зависимости от текущего режима работы имела бы два различных набора свойств и выполняла бы различные функции. Вы можете разместить в нужном месте две кнопки друг на друге (пусть они имеют имена **Button1** и **Button2**), задать каждой нужные свойства и для каждой описать соответствующие обработчики событий. Тогда для смены этих кнопок вы в соответствующих местах кода программы пишете операторы

```
Button1->BringToFront();
```

или

```
Button2->BringToFront();
```

и пользователь будет видеть то одну, то другую из этих кнопок.

2. Пусть в приложении в одном и том же месте формы друг на друге расположены две панели: **Panel1** и на ней **Panel2**, содержащие какие-то управляющие компоненты для разных режимов работы. **Panel2** расположена на **Panel1**, которая является, таким образом, ее родителем. В обработчик события формы **OnCreate** можно вставить операторы:

```
Panel2->Parent = Form1;  
Panel2->BoundsRect = Panel1->BoundsRect;  
Panel1->BringToFront();
```

Первый оператор делает родительским компонентом панели **Panel2** форму **Form1**. Вторым оператор задает панели **Panel2** то же местоположение и размеры, которые имеет панель **Panel1**. Последнее необходимо, поскольку при проектировании ее координаты соответствовали координатному пространству контейнера — клиентской области панели **Panel1**. А теперь ее родитель сменился на форму, и надо ее расположить в том же месте формы, в котором расположена **Panel1**. Третий оператор перемещает наверх форму **Panel1**.

Приведенный код можно сократить, и убрать из него первый оператор, если в процессе проектирования размещать панель **Panel2** не на панели **Panel1**, а в каком-то другом месте непосредственно на форме (это, кстати, много удобнее с точки зрения проектирования каждой панели). Тогда в обработчике события формы **OnCreate** достаточно двух операторов:

```
Panel2->BoundsRect = Panel1->BoundsRect;  
Panel1->BringToFront();
```

изменяющих положение **Panel2** и перемещающих наверх форму **Panel1**.

В результате работы приведенных операторов в момент создания формы на ней будет видна панель **Panel1**. В момент, когда ее надо заменить на **Panel2**, можно выполнить оператор:

```
Panel2->BringToFront();
```

выносящий наверх вторую панель. Когда надо вернуть на экран изображение **Panel1**, можно выполнить операторы:

```
Panel1->BringToFront();
```

Аналогичный пример приведен в разделе **Visible**, но использование метода **BringToFront** делает его более компактным.

BrushCopy

Копирует часть изображения битовой матрицы на данную канву, заменяя указанный цвет в изображении на значение, установленное для кисти канвы

Класс *TCanvas*

Прототип

```
void __fastcall BrushCopy(const Windows::TRect &Dest,  
                           TBitmap* Bitmap,  
                           const Windows::TRect &Source,  
                           TColor Color);
```

Описание

Метод **BrushCopy** копирует часть изображения битовой матрицы компонента **Bitmap** на данную канву, заменяя указанный цвет **Color** в изображении на значение, установленное для кисти канвы **Brush**. Параметр **Source** указывает копируемую прямоугольную область в источнике изображения **Bitmap**. Параметр **Dest** указывает прямоугольную область на канве, в которую производится копирование.

Замена цвета делает изображение как бы частично прозрачным, если в параметре **Color** указать цвет фона изображения, а в параметре **Color** кисти **Brush** канвы указать цвет фона канвы.

Пример

Оператор

```
Form1->Canvas->BrushCopy(Rect(10,10,100,100),  
                          Bitmap1, Rect(10,10,100,100), clBlack);
```

копирует прямоугольную область с координатами углов (10, 10) и (100, 100) из компонента **Bitmap1** в аналогичную область канвы формы **Form1** и заменяет в изображении черный цвет на цвет, установленный в свойстве **Form1->Canvas->Brush->Color**.

CanFocus

Определяет, может ли компонент получать сообщения пользователя

Класс *TWinControl*

Прототип

```
bool __fastcall CanFocus(void);
```

Описание

Метод **CanFocus** определяет, может ли компонент получать сообщения пользователя, т.е. может ли он получать фокус. Функция возвращает **true**, если у компонента и всех его родителей свойства **Visible** и **Enabled** установлены в **true**. В противном случае возвращается **false**.

ChangeScale

Изменяет масштаб компонента и его дочерних компонентов

Классы *TControl*, *TCustomForm*, *TScrollingWinControl*, *TWinControl*

Прототип

```
DYNAMIC void __fastcall ChangeScale(int M, int D);
```

Описание

Метод **ChangeScale** используется для изменения масштаба компонента. Масштабируются такие свойства компонента, как **Top** и **Left**, определяющие его местоположение (этим **ChangeScale** отличается от метода **ScaleBy**, который не затрагивает **Top** и **Left**), а также **Width** и **Height**, определяющие его размер. В классах, производных от **TControl**, в частности, в **TWinControl**, масштабируются также все компоненты, принадлежащие данному компоненту, и их шрифты.

Параметры **M** and **D** определяют соответственно множитель и делитель масштаба. Например, чтобы уменьшить размеры до 75% начального значения, можно задать **M** равным 75, а **D** равным 100 (75/100). То же самое можно сделать, задав **M**=3 и **D**=4 (3/4). Если вы хотите увеличить размер на 1/3, то можно задать **M**=133 и **D**=100 (133/100) или **M**=4 и **D**=3 (4/3).

Chord

Рисует заполненную замкнутую фигуру, ограниченную дугой окружности или эллипса и хордой

Класс *TCanvas*

Прототип

```
void __fastcall Chord(int X1, int Y1, int X2, int Y2,
                     int X3, int Y3, int X4, int Y4);
```

Описание

Метод **Chord** рисует замкнутую фигуру: дугу окружности или эллипса, замкнутую хордой, с помощью текущих параметров пера **Pen**. Фигура заполняется текущим значением **Brush**. Точки (**X1**, **Y1**) и (**X2**, **Y2**) определяют прямоугольник, описывающий эллипс. Начальная точка дуги определяется пересечением эллипса с прямой, проходящей через его центр и точку (**X3**, **Y3**). Конечная точка дуги определяется пересечением эллипса с прямой, проходящей через его центр и точку (**X4**, **Y4**). Дуга рисуется против часовой стрелки от начальной до конечной точки. Хорда соединяет точки (**X3**, **Y3**) и (**X4**, **Y4**).

В Windows 95/98 суммы **X1 + X2**, **Y1 + Y2** и **X1 + X2 + Y1 + Y2** не должны превышать 32768.

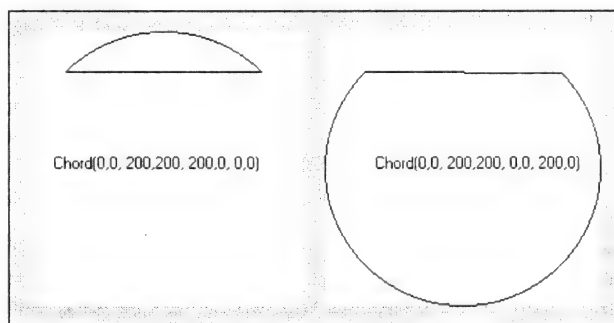
В Windows NT направление рисования дуги можно изменить на направление по часовой стрелке вызовом функции **SetArcDirection**.

Примеры

Операторы

```
Image1->Canvas->Chord(0,0, 200,200, 200,0, 0,0);
Image2->Canvas->Chord(0,0, 200,200, 0,0, 200,0);
```

дают результат, показанный на рисунке.



ClassName

Возвращает имя типа объекта

Класс *TObject*

Определение

```
static ShortString __fastcall ClassName(TClass cls);
```

Описание

Метод **ClassName** возвращает имя действительного типа объекта. Например, переменная типа класса-предка может ссылаться на экземпляр любого типа-потомка. В этом случае **ClassName** возвращает имя реального типа объекта, а не того, которое было объявлено для этой ссылки. Например, оператор

```
catch(Exception& E)
{
    ShowMessage("Возникло исключение "+E.ClassName());
}
```

перехватит все исключения и отобразит сообщение с именем действительно сгенерированного исключения. Впрочем, в данной ситуации лучше применить оператор

```
catch(Exception& E)
{
    ShowMessage("Возникло исключение "+E.Message);
}
```

который отобразит пользователю не класс исключения, а сообщение этого класса.

Clear

Очистка списков

Классы *TClipboard*, *TList*, *TStringList*, *TStrings*, *TComboBox*, *TDBCombobox*, *TDBEdit*, *TDBListBox*, *TDBMemo*, *TDirectoryListBox*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TListBox*, *TMaskEdit*, *TMemo*, *TOutline* и ряд других

Определение

```
DYNAMIC void __fastcall Clear(void);
```

Описание

Для перечисленных выше объектов и компонентов процедура **Clear** удаляет все элементы списков или весь текст. Для некоторых других объектов аналогичная процедура действует несколько иначе.

Для объекта **Clipboard** процедура **Clear** удаляет все содержимое буфера Clipboard. Впрочем, то же самое происходит автоматически при каждом обновлении содержимого буфера (при выполнении операций вырезать и копировать — Cut и Copy).

ClientToScreen

Преобразует координаты клиентской области в координаты экрана

Класс *TControl*

Определение

```
struct TPoint
{
    int x;
    int y;
};
```

```
Windows::TPoint __fastcall ClientToScreen(
                                const Windows::TPoint &Point);
```

Описание

Метод **ClientToScreen** преобразует координаты точки в системе координат клиентской области компонента (начало координат — левый верхний угол клиентской области) в систему координат экрана (начало координат — левый верхний угол экрана).

Совместно с обратной функцией **ScreenToClient** метод может использоваться для пересчета координат точки экрана из системы координат клиентской области одного компонента в систему координат клиентской области другого компонента.

Пример

```
P = Comp2->ScreenToClient(Comp1->ClientToScreen(P));
```

Оператор пересчитывает координату точки экрана **P** из системы координат компонента **Comp1** в систему координат компонента **Comp2**.

ContainsControl

Определяет, является ли указанный компонент прямым или косвенным наследником данного оконного компонента

Класс *TWinControl*

Прототип

```
bool __fastcall ContainsControl(TControl* Control);
```

Описание

Метод **ContainsControl** позволяет определить, является ли компонент, указанный параметром **Control**, наследником данного оконного компонента. Метод возвращает **true** не только, если в свойстве **Controls** компонента **Control** указан в качестве родителя данный компонент, но и если он является прямым или косвенным потомком какого-то из дочерних компонентов данного оконного компонента.

ControlAtPos

Определяет, какой дочерний компонент имеется в указанной позиции

Класс *TWinControl*

Прототип

```
struct TPoint
{
    int x;
    int y;
};
```

```
TControl* __fastcall ControlAtPos(const Windows::TPoint &Pos,
                                bool AllowDisabled);
```


Описание

Метод **ControlAtPos** позволяет определить, какой дочерний компонент данного оконного элемента имеется в позиции с координатами, указанными параметром **Pos**. Возвращается только непосредственно дочерний компонент, т.е. такой, в чьем свойстве **Parent** указан данный оконный элемент и который поэтому входит в список дочерних компонентов, содержащийся в свойстве **Controls** оконного элемента.

Позиция **Pos** может находиться в любом месте внутри дочернего компонента. Если заданная позиция не соответствует никакому дочернему компоненту, то функция **ControlAtPos** возвращает **NULL**.

Параметр **AllowDisabled** определяет, учитываются ли при поиске компоненты, которые находятся в недоступном состоянии.

CopyRect

Копирует часть изображения с другой канвы на данную

Класс *TCanvas*

Прототип

```
void __fastcall CopyRect(const Windows::TRect &Dest,
                        TCanvas* Canvas, const Windows::TRect &Source);
```

Описание

Метод **CopyRect** переносит указанную параметром **Source** область изображения в канве источника изображения **Canvas** в указанную параметром **Dest** область данного объекта **TCanvas**. Копирование производится в режиме, установленном свойством **CopyMode**.

Пример

Оператор

```
Image1->Canvas->CopyRect(MyRect2, Bitmap->Canvas, MyRect1);
```

копирует на канву компонента **Image1** в область **MyRect2** изображение из области **MyRect1** канвы компонента **Bitmap**.

DbClick

См. раздел **TControl**.

Delete

Удаление элемента с указанным индексом из списка

Классы *TList*, *TStringList*, *TStrings*, *TMenuItem*

Объявление

```
void __fastcall Delete(int Index);
```

Описание

Процедура **Delete** удаляет из списка элемент с указанным индексом. Во всех случаях индексы считаются, начиная с 0 (0 — индекс первого элемента).

Из списка строк строка удаляется вместе со ссылкой на связанный с ней объект.

Удаление элемента меню ведет к удалению и связанного с ним подменю (если таковое имеется).

После удаления элемента список перестраивается. Это надо учитывать при удалении элементов в цикле, как указано в приведенном примере.

Пример

```
for (I = 0; I <= 1; I++)
    List->Delete(0);
```

В этом примере удаляются два первых элемента списка **List**. Неверно было бы написать:

```
for (I = 0; I <= 1; I++)  
    List->Delete(I);
```

Так как после первого удаления список перестроится, то элементом с индексом 1 станет тот, который ранее имел индекс 2. Таким образом, в результате оказались бы удаленными элементы, имевшие в первоначальном списке индексы 0 и 2.

DisableAlign

Временно запрещает выравнивание дочерних компонентов

Класс *TWinControl*

Определение

```
void __fastcall DisableAlign(void);
```

Описание

Свойство **DisableAlign** временно запрещает выравнивание дочерних компонентов внутри оконного элемента - изменение их свойств **Align** не приводит к их перестроению. Метод применяется совместно с методом **EnableAlign**, отменяющим действие **DisableAlign** и разрешающим выравнивание.

Эти методы целесообразно использовать, если проводится перестроение ряда дочерних компонентов, например, при чтении их из файла формы, при масштабировании, при изменении взаимного расположения.

Каждому вызову **DisableAlign** должен соответствовать вызов **EnableAlign**. Очередной вызов **DisableAlign** увеличивает на единицу число запретов выравнивания, а вызов **EnableAlign** уменьшает это число. Как только при очередном вызове **EnableAlign** число запретов станет равным нулю, произойдет выравнивание. Это производится вызовом метода **Realign**.

Пример

```
Form1->DisableAlign()  
    <операторы перестроения дочерних компонентов>  
Form1->EnableAlign()
```

Dormant

Создает изображение битовой матрицы в памяти, чтобы освободить дескриптор матрицы и сэкономить ресурсы

Класс *TBitmap*

Прототип

```
void __fastcall Dormant(void);
```

Описание

Использование метода **Dormant** сокращает ресурсы GDI, используемые в приложении. Метод создает изображение матрицы в памяти, используя объект потока памяти. В дальнейшем через свойство **Handle** можно освободить связанный с матрицей HBITMAP.

Дополнительную экономию памяти можно получить методом **FreeImage**, освобождающим память, занятую кэшированием изображения. Но этот метод может приводить к потере глубины цвета.

Пример

```
void __fastcall TForm1::Button1Click(TObject *Sender)  
{  
    Graphics::TBitmap *BitMap1 = new Graphics::TBitmap();
```

```

Graphics::TBitmap *BitMap2 = new Graphics::TBitmap();

try
{
    // загрузка изображения из файла в BitMap1
    BitMap1->LoadFromFile("...");
    // копирование в BitMap2 из BitMap1
    BitMap2->Assign(BitMap1);
    // освобождение ресурсов GDI
    BitMap2->Dormant();
    // освобождение памяти, изображение не теряется
    BitMap2->FreeImage();
    // изображение из BitMap2 рисуется на канве
    Canvas->Draw(20,20,BitMap2);
    // устанавливается монохромный режим BitMap2
    // BitMap2->Monochrome = true;
    // рисуется монохромный вариант
    Canvas->Draw(250,20,BitMap2);
    // теперь изображение действительно теряется
    BitMap2->ReleaseHandle();
}
catch (...)
{
    MessageBeep(0);
}
delete BitMap1;
delete BitMap2;
}

```

Draw

Рисует графическое изображение в указанную позицию канвы

Класс *TCanvas*

Прототип

```
void __fastcall Draw(int X, int Y, TGraphic* Graphic);
```

Описание

Метод **Draw** рисует изображение, содержащееся в объекте, указанном параметром **Graphic**, сохраняя исходный размер изображения в его источнике и перенося изображение в область канвы объекта. Верхний левый угол этой области определяется параметрами **X** и **Y**. Источник изображения может быть битовой матрицей, пиктограммой или метафайлом. Если источник — объект типа **TBitmap**, то перенос изображения производится в режиме, установленном свойством канвы **CopyMode**.

Пример

Оператор

```
Image1->Canvas->Draw(10,10, Bitmap1);
```

рисует на канве компонента **Image1** изображение из компонента **Bitmap1** в область с координатами левого верхнего угла (10, 10).

DrawFocusRect

Рисует изображение прямоугольника в виде, используемом для отображения рамки фокуса, операцией **xor**

Класс *TCanvas*

Прототип

```
void __fastcall DrawFocusRect(const Windows::TRect &Rect);
```

Описание

Метод **DrawFocusRect** рисует на канве в области **Rect** изображение прямоугольника в виде, используемом обычно для отображения рамки фокуса, т.е. точками. При рисовании используется операция **xor**, что позволяет удалить изображение прямоугольника его повторной прорисовкой.

Пример

Следующая совокупность обработчиков событий, связанных с мышью, рисует на канве компонента **Imag1** прямоугольную рамку размером 10 на 10 вокруг курсора и перетаскивает ее при перемещении мыши с нажатой кнопкой:

```
int X0,Y0;
bool drag = false;

void __fastcall TForm1::Imag1MouseDown(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
    // рисование рамки
    Imag1->Canvas->DrawFocusRect(Rect(X-5,Y-5,X+5,Y+5));
    // запоминание координат курсора
    X0 = X;
    Y0 = Y;
    // включение флажка режима перемещения рамки
    drag = true;
}
//-----
void __fastcall TForm1::Imag1MouseMove(TObject *Sender,
    TShiftState Shift, int X, int Y)
{
    if( !drag) return;
    // стирание рамки
    Imag1->Canvas->DrawFocusRect(Rect(X0-5,Y0-5,X0+5,Y0+5));
    // рисование рамки
    Imag1->Canvas->DrawFocusRect(Rect(X-5,Y-5,X+5,Y+5));
    // запоминание координат курсора
    X0 = X;
    Y0 = Y;
}
//-----
void __fastcall TForm1::Imag1MouseUp(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
    if( !drag) return;
    // стирание рамки
    Imag1->Canvas->DrawFocusRect(Rect(X0-5,Y0-5,X0+5,Y0+5));
    // выключение флажка режима перемещения рамки
    drag = false;
}
```

При нажатии кнопки мыши рисуется первая рамка, запоминаются координаты курсора и включается режим перемещения рамки (переменная **drag = true**). При перемещении мыши в режиме перемещения рамки стирается прежняя рамка, рисуется рамка в новой позиции и запоминаются новые координаты курсора. При отпускании кнопки мыши стирается рамка и выключается режим перемещения рамки.

Ellipse

Рисует заполненную окружность или эллипс

Класс *TCanvas*

Прототип

```
void __fastcall Ellipse(int X1, int Y1, int X2, int Y2);
```

Описание

Метод **Ellipse** рисует окружность или эллипс с помощью текущих параметров пера **Pen**. Фигура заполняется текущим значением **Brush**. Точки (X1, Y1) и (X2, Y2) определяют прямоугольник, описывающий эллипс.

В Windows 95/98 суммы $X1 + X2$, $Y1 + Y2$ и $X1 + X2 + Y1 + Y2$ не должны превышать 32768.

Пример**Оператор**

```
Image1->Canvas->Brush->Color = clRed;
Image1->Canvas->Brush->Style = bsDiagCross;
Image1->Canvas->Ellipse(0, 0, Image1->Width, Image1->Height);
```

рисует эллипс, вписанный в компонент **Image1** и заполненный красной штриховкой.

EnableAlign

См. **DisableAlign**

Exchange

Меняет позиции двух элементов списка

Классы *TList*, *TStringList*, *Tstrings*

Прототипы

Для **TList**:

```
void __fastcall Exchange(int Index1, int Index2);
```

для **TStrings** и **TStringList**:

```
virtual void __fastcall Exchange(int Index1, int Index2);
```

Описание

При вызове **Exchange** два элемента списка с позициями **Index1** и **Index2** обмениваются местами. Индексы позиций начинаются с 0 (0 — первый элемент).

Если в списках строк со строками связанными объекты, они остаются связанными с теми же строками в их новых позициях.

Не применяйте метод **Exchange** к отсортированным спискам **TStringList**, за исключением случая взаимного перемещения двух одинаковых строк, связанных с разными объектами. Дело в том, что метод **Exchange** не проверяет условий сортировки и может нарушить упорядоченность списка.

Пример

Пусть список **List** типа **TList** содержит указатели на целые числа. Тогда приведенная ниже программа перемещает на первое место указатель на минимальное число (это один шаг пузырьковой сортировки).

```
for(int i = 1; i < List->Count; i++)
    if(*((int *)List->Items[0]) > *((int *)List->Items[i]))
        List->Exchange(0, i);
```

Полностью пузырьковая сортировка списка указателей на целые числа реализуется вложенными циклами:

```
for(int j = 0; j < List->Count-1; j++)
    for(int i = j; i < List->Count; i++)
        if (*((int *)List->Items[j]) > *((int *)List->Items[i]))
            List->Exchange(j, i);
```

Expand

Увеличивает емкость списка типа **TList**

Класс *TList*

Прототип

```
TList* __fastcall Expand(void);
```

Описание

Метод **Expand** увеличивает емкость списка типа **TList**, выделяя память для быстрого размещения новых элементов. Тем самым экономится время при добавлении в дальнейшем новых элементов списка.

Поскольку функция возвращает расширенный список, то ее можно также использовать для создания дубликата имеющегося списка.

Если список не заполнен, т.е. количество элементов **Count** меньше емкости списка **Capacity**, то список не расширяется. Если же **Count** = **Capacity**, то функция **Expand** увеличивает емкость **Capacity** согласно следующему алгоритму. Если значение **Capacity** меньше 4, то **Capacity** увеличивается на 4. Если значение **Capacity** больше 4, но меньше 9, то **Capacity** увеличивается на 8. Если значение **Capacity** больше 8, то **Capacity** увеличивается на 16.

Примеры

1. `List->Expand();`

Если список **List** заполнен не до конца, он остается неизменным. Если же он заполнен, то его емкость расширяется.

2. `List1 = List->Expand();`

Создается объект **List1**, являющийся копией списка **List**. Если исходный список был заполнен, то оба списка **List** и **List1** оказываются расширенными.

FillRect

Заполняет указанный прямоугольник канвы, используя текущее значение **Brush**

Класс *TCanvas*

Прототип

```
void __fastcall FillRect(const Windows::TRect &Rect);
```

Описание

Метод **FillRect** заполняет прямоугольник канвы, указанный параметром **Rect**, используя текущее значение кисти **Brush**. Заполняемая область включает верхнюю и левую стороны прямоугольника, но не включает правую и нижнюю стороны. При использовании **FillRect** параметр **Rect** часто задается функцией **Rect** (см. раздел 15.3.3 главы 15).

Пример

Оператор

```
Image1->Canvas->FillRect(Rect(0,0, Image1->Width,  
                             Image1->Height));
```

очищает всю канву компонента **Image1**, заполняя ее фоном, если он установлен в свойстве **Brush**.

FindNextControl

Возвращает следующий в последовательности табуляции оконный дочерний компонент

Класс *TWinControl*

Определение

```
TWinControl* __fastcall FindNextControl(
    TWinControl* CurControl, bool GoForward,
    bool CheckTabStop, bool CheckParent);
```

Описание

Метод **FindNextControl** находит и возвращает следующий за указанным в параметре **CurControl** дочерний оконный компонент в соответствии с последовательностью табуляции. Если **CurControl** не является дочерним компонентом данного оконного элемента, то возвращается компонент, первый в последовательности табуляции. То же самое происходит, если **CurControl** является последним компонентом в последовательности табуляции.

Параметр **GoForward** определяет направление поиска. Если он равен **true**, то поиск проводится вперед и возвращается компонент, следующий за **CurControl**. Если же параметр **GoForward** равен **false**, то возвращается предшествующий компонент.

Параметры **CheckTabStop** и **CheckParent** определяют условия поиска. Если **CheckTabStop** равен **true**, то просматриваются только компоненты, в которых свойство **TabStop** установлено в **true**. При **CheckTabStop** равном **false** значение **TabStop** не принимается во внимание. Если параметр **CheckParent** равен **true**, то просматриваются только компоненты, в свойстве **Parent** которых указан данный оконный элемент, т.е. просматриваются только прямые потомки. Если **CheckParent** равен **false**, то просматриваются все, даже косвенные потомки данного элемента.

Метод **FindNextControl** вызывает метод **GetTabOrderList** и из полученного таким способом списка черпает последовательность компонентов.

Примеры

```
TWinControl *obj;
for(int i = 0; i < Form1->ControlCount; i++)
{
    obj = Form1->FindNextControl(obj,true, true, true);
    ...
}
```

В этом примере переменная **obj** поочередно принимает значение всех прямых наследников формы **Form1**, включенных в последовательность табуляции, т.е. имеющих свойство **TabStop** равным **true**. Например, в эту последовательность войдут окна редактирования, кнопки, панели, расположенные непосредственно на форме и имеющие **TabStop** равным **true**, но не войдут кнопки и окна редактирования, расположенные на панелях.

Если в приведенном операторе изменить параметр **CheckParent** на **false**:

```
obj = Form1->FindNextControl(obj,true, true, false);
```

то в последовательность войдут и не прямые наследники, имеющие **TabStop** равным **true**, в частности, компоненты, содержащиеся в панелях, расположенных на форме, причем независимо от значения **TabStop** этих панелей.

Если в приведенном операторе изменить параметр **CheckTabStop** на **false**:

```
obj = Form1->FindNextControl(obj,true, false, false);
```

то в последовательность войдут компоненты, независимо от значения их свойства **TabStop**.

FloodFill

Закрашивает текущей кистью замкнутую область канвы, определенную указанным цветом

Класс *TCanvas*

Прототип

```
enum TFillStyle {fsSurface, fsBorder};  
void __fastcall FloodFill(int X, int Y, TColor Color,  
                          TFillStyle FillStyle);
```

Описание

Метод **FloodFill** закрашивает текущей кистью **Brush** замкнутую область канвы, определенную цветом и начальной точкой закрашивания (**X**, **Y**). Точка с координатами **X** и **Y** является произвольной внутренней точкой заполняемой области, которая может иметь произвольную форму. Граница этой области определяется сочетанием параметров **Color** и **FillStyle**. Параметр **Color** типа **TColor** указывает цвет, который используется при определении границы закрашиваемой области, а параметр **FillStyle** определяет, как именно по этому цвету определяется граница. Если **FillStyle** = **fsSurface**, то заполняется область, окрашенная цветом **Color**, а на других цветах метод останавливается. Если **FillStyle** = **fsBorder**, то наоборот, заполняется область окрашенная любыми цветами, не равными **Color**, а на цвете **Color** метод останавливается.

Примеры

1.

```
Image1->Canvas->Brush->Color = clWhite;  
Image1->Canvas->FloodFill(X, Y,  
                          Image1->Canvas->Pixels[X][Y], fsSurface);
```

Приведенные операторы закрашивают белым цветом на канве компонента **Image1** все пиксели, прилегающие к пикселю с координатами (**X**, **Y**) и имеющие тот же цвет, что и этот пиксель. Например, если вы вставите эти операторы в обработчик щелчка **OnClick** компонента **Image1**, то пикселем, определяющим закраску, будет пиксель той точки, в которой пользователь щелкнул на изображении.

2.

```
Image1->Canvas->Brush->Color = clWhite;  
Image1->Canvas->FloodFill(X, Y, clBlack, fsBorder);
```

Приведенные операторы закрашивают белым цветом на канве компонента **Image1** все пиксели, прилегающие к пикселю с координатами (**X**, **Y**) и имеющие цвет, отличный от черного. При достижении черной границы области за-краска останавливается.

Focused

Определяет, находится ли оконный элемент в фокусе

Класс *TWinControl*

Определение

```
DYNAMIC bool __fastcall Focused(void);
```

Описание

Метод **Focused** определяет, является ли оконный элемент активным, т.е. находится ли он в фокусе. Возвращает **true**, если элемент находится в фокусе, и **false** — если элемент не в фокусе и пользователь в данный момент не может с ним взаимодействовать.

FrameRect

Рисует на канве текущей кистью прямоугольную рамку

Класс *TCanvas*

Прототип

```
void __fastcall FrameRect(const Windows::TRect &Rect);
```

Описание

Метод **FrameRect** рисует на канве прямоугольную рамку вокруг области **Rect**, используя установку текущей кисти **Brush**. Толщина рамки — 1 пиксель. Область внутри рамки кистью не заполняется. Отличается от метода **Rectangle** тем, что рамка рисуется цветом кисти (в методе **Rectangle** — цветом пера **Pen**) и область не закрашивается (в методе **Rectangle** закрашивается).

Пример

Оператор

```
Image1->Canvas->Brush->Color = clBlack;  
Image1->Canvas->FrameRect(Rect(10,10,100,100));
```

рисует на канве компонента **Image1** черную рамку.

Free

Вызывает деструктор объекта и освобождает память

Класс *TObject*

Прототип

```
__fastcall Free();
```

Описание

Функцию **Free** не следует применять непосредственно для освобождения памяти, динамически выделенной под объект, который уже не нужен для дальнейшей работы программы. Вместо этого следует использовать ключевое слово **delete**, по которому автоматически вызывается функция **Free**.

Функцию **Free** проверяет, не была ли ранее уже освобождена выделенная под объект память и вообще был ли данный объект создан (не равен ли указатель на объект **NULL**). После этого вызывается деструктор данного объекта.

GetTabOrderList

Строит список дочерних оконных компонентов в последовательности табуляции

Класс *TWinControl*

Определение

```
DYNAMIC void __fastcall GetTabOrderList(Classes::TList* List);
```

Описание

Метод **GetTabOrderList** строит список **List** типа **TList** дочерних оконных компонентов в последовательности табуляции. В список входят не только непосредственные потомки данного элемента, но и косвенные потомки, включенные в дочерние контейнеры. При этом не обращается внимание на значение свойства **TabStop** включаемых компонентов.

Метод **GetTabOrderList** вызывается методом **FindNextControl**, определяющим последующий или предшествующий компонент в списке табуляции.

Пример

Приведенные ниже операторы обеспечивают поочередный доступ ко всем дочерним оконным компонентам в последовательности табуляции.

```
TWinControl *obj;  
TList *List = new TList;  
...  
GetTabOrderList(List);  
...  
for(int i = 0; i < List->Count; i++)  
{  
    // Поочередный доступ к объектам  
    obj = (TWinControl *)List->Items[i];  
    ...  
}
```

HandleAllocated

Проверяет наличие дескриптора окна компонента

Класс *TWinControl*

Определение

```
bool __fastcall HandleAllocated(void);
```

Описание

Метод **HandleAllocated** используется для определения, имеется ли дескриптор окна у данного элемента. Если дескриптор имеется, то возвращается **true**.

Непосредственная проверка свойства **Handle** приводит к тому, что даже если дескриптора не было, он создается. Применение **HandleAllocated** позволяет определить наличие дескриптора без этого побочного эффекта.

HandleNeeded

Создает дескриптор окна, если до этого он не существовал

Класс *TWinControl*

Определение

```
void __fastcall HandleNeeded(void);
```

Описание

Метод **HandleNeeded** создает дескриптор окна, если до этого его не было. При создании дескриптора он прежде всего вызывает метод **CreateHandle** родительского элемента, а уже затем создает дескриптор данного элемента.

Hide

Делает компонент невидимым

Класс *TControl*

Определение

```
void __fastcall Hide(void);
```

Описание

Метод **Hide** делает компонент невидимым, задавая значение **False** его свойству **Visible**. Если компонент является контейнером для других компонентов, то эти дочерние компоненты также делаются невидимыми.

Хотя компонент становится невидимым, его свойства и методы остаются доступными.

IndexOf

Определение первого вхождения в список заданного элемента

Классы *TList*, *TStringList*, *TStrings*

ПрототипыДля **TList**:

```
int __fastcall IndexOf(void * Item);
```

для **TStrings** и **TStringList**:

```
virtual int __fastcall IndexOf(const System::AnsiString S);
```

Описание

Вызов **IndexOf** возвращает индекс первого вхождения в массив списка заданного элемента (указателя **Item** для **TList** или строки **S** для **TStringList** и **TStrings**). Индексация начинается с 0 (0 — первый элемент массива). Если заданного элемента в списке нет, возвращается -1.

Пример

В приведенном ниже примере определяется, есть ли в списке сотрудник, фамилия которого задана пользователем в окне **Edit1**.

```
TStringList *LPerson = new TStringList;
...
if (LPerson->IndexOf(Edit1->Text) < 0)
    ShowMessage("Сотрудника " + Edit1->Text +
               " в списке нет");
```

Insert

Процедура вставляет элемент в список в заданную позицию

Классы *TList*, *TStringList*, *TStrings***Прототипы**Для **TList**:

```
void __fastcall Insert(int Index, void * Item);
```

для **TStrings** и **TStringList**:

```
virtual void __fastcall Insert(int Index,
                             const System::AnsiString S);
```

Описание

Процедура **Insert** вставляет элемент (указатель **Item** или строку **S**) в список в позицию, индекс которой задан параметром **Index**. Если задан **Index** = 0, элемент вставляется в первую позицию. При вставке элемента все имеющиеся в списке элементы с индексами, равными и большими **Index**, сдвигаются, т.е. их индексы увеличиваются на 1.

Если список сортирован, то вызов **Insert** приводит к генерации исключения **EListError**. Для сортированных списков следует использовать метод **Add**.

Если в списки **TStringList** и **TStrings** надо вставить строку, связанную с объектом, то вместо метода **Insert** следует использовать метод **InsertObject**.

Invalidate

Сообщает Windows о необходимости полностью перерисовать компонент после того, как будут обработаны другие важные сообщения Windows

Класс *TControl***Определение**

```
virtual void __fastcall Invalidate(void);
```

Описание

Метод **Invalidate** надо вызывать, когда требуется полностью перерисовать компонент. Если более одной области компонента требует перерисовки, вызов ме-

тогда **Invalidate** приводит к его полной перерисовке, что позволяет избежать мерцания изображения. Многократный вызов **Invalidate** до действительной перерисовки компонента не приводит к потере эффективности работы.

LineTo

Рисует на канве прямую линию, начинающуюся с текущей позиции пера и кончающуюся указанной точкой

Класс *TCanvas*

Прототип

```
void __fastcall LineTo(int X, int Y);
```

Описание

Метод **LineTo** рисует на канве прямую линию, начинающуюся с текущей позиции пера **PenPos** и кончающуюся точкой (X, Y), исключая саму точку (X, Y). Текущая позиция пера **PenPos** перемещается в точку (X, Y). При рисовании используются текущие установки пера **Pen**.

Пример

Операторы

```
Image1->Canvas->MoveTo(X1,Y1);
Image1->Canvas->LineTo(X2,Y2);
Image1->Canvas->LineTo(X3,Y3);
```

рисуют кусочно-ломаную прямую, соединяющую точки (X1,Y1), (X2,Y2) и (X3,Y3).

LoadFromClipboardFormat

Загружает изображение из буфера обмена в формате Clipboard

Класс *TGraphic, TBitmap, TIcon, TMetafile, TPicture*

Прототип

```
virtual void __fastcall LoadFromClipboardFormat(
    Word AFormat, int AData, HPALETTE APalette);
```

Описание

Метод загружает изображение в графический объект в указанном формате Clipboard. Если формат **AFormat** найден среди зарегистрированных, то **AData** и **APalette** передаются для загрузки изображения. Стандартно зарегистрированные форматы: **CF_BITMAP** для битовых карт и **CF_METAFILEPICT** для метафайлов. Значение **AData** может быть указано методом **GetAsHandle** объекта типа **TClipboard**. При этом надо не забыть включить в приложение директиву

```
#include <vcl\Clipbrd.hpp>
```

Формат для нового типа графического объекта предварительно должен быть зарегистрирован методом **RegisterClipboardFormat**.

Если в буфере обмена находится не тот тип данных, который ожидается, то генерируется исключение **EInvalidGraphic**.

Пример

```
#include <vcl\Clipbrd.hpp>
...
if (Clipboard()->HasFormat(CF_BITMAP))
{
    try
    {
        Image1->Picture->Bitmap->LoadFromClipboardFormat(CF_BITMAP,
            Clipboard()->GetAsHandle(CF_BITMAP), 0);
    }
}
```

```

    }
    catch (...)
    {
        ShowMessage("Загрузка изображения невозможна");
    }
}
else
    ShowMessage("В буфере не точечное изображение");

```

Приведенный код загружает изображение из буфера обмена в формате битовой карты в компонент **Image1**.

LoadFromFile — метод графических объектов

Загружает изображение, хранящееся в файле

Классы *TGraphic*, *TPicture*

Прототип

```
virtual void __fastcall LoadFromFile(const AnsiString FileName);
```

Описание

Метод **LoadFromFile** читает файл **FileName** и загружает его в графический объект.

Если формат графического файла не зарегистрирован, или не соответствует типу графического объекта, то генерируется исключение **EInvalidGraphic**.

Пример

```

if (OpenPictureDialog1->Execute())
    Image1->Picture->LoadFromFile(OpenPictureDialog1->FileName);

```

Этот оператор открывает диалог **OpenPictureDialog1**, позволяющий пользователю выбрать файл, и загружает изображение из файла в компонент **Image1**.

LoadFromResourceID

Загружает битовую карту из файла ресурсов по указанному идентификатору

Класс *TBitmap*

Прототип

```
void __fastcall LoadFromResourceID(int Instance, int ResID);
```

Описание

Метод **LoadFromResourceID** загружает битовую карту из файла ресурсов выполняемого модуля. Загружаемая карта указывается идентификатором **ResID**.

LoadFromResourceName

Загружает битовую карту из файла ресурсов по указанному имени

Класс *TBitmap*

Прототип

```
void __fastcall LoadFromResourceName(int Instance,
                                     const AnsiString ResName);
```

Описание

Метод **LoadFromResourceName** загружает битовую карту из файла ресурсов выполняемого модуля. Загружаемая карта указывается именем **ResName**. Метод поддерживает изображения с 256 цветами.

LoadFromStream

Загружает графическое изображение из указанного потока

Класс *TGraphic*

Прототип

```
virtual void __fastcall LoadFromStream(Classes::TStream* Stream);
```

Описание

Метод **LoadFromStream** читает из потока **Stream** графический объект. Используется, например, для загрузки изображения из объекта **TBlobStream**, чтобы прочитать графическое поле в наборе данных.

Lock

Блокирует канву, не разрешая другим нитям многопоточного приложения рисовать на ней

Класс *TCanvas*

Прототип

```
void __fastcall Lock(void);
```

Описание

Метод **Lock** блокирует данную канву, не разрешая другим нитям многопоточного приложения рисовать на ней. Канва остается блокированной до снятия блокады вызовом метода **Unlock**. Если имеются вложенные вызовы **Lock**, то они увеличивают свойство **LockCount**, фиксирующее количество блокировок. Канва будет оставаться блокированной, пока не будет снята последняя блокировка.

Если нежелательна вложенная многократная блокировка, лучше использовать метод **TryLock**.

Поскольку блокировка не дает другим нитям рисовать на канве, производительность работы приложения может за счет этого снизиться. Так что не надо злоупотреблять блокировками. Их следует применять только тогда, когда есть вероятность нежелательных наложений операций, выполняемых в разных нитях приложения с несколькими потоками.

Move

Меняет текущую позицию элемента в списке на заданную

Модуль *System*

Классы *TList*, *TStringList*, *Tstrings*

Прототип

```
void __fastcall Move(int CurIndex, int NewIndex);
```

Описание

Процедура **Move** меняет текущую позицию элемента в списке, индекс которого задан параметром **CurIndex**, на позицию с индексом, заданным параметром **NewIndex** (индексы начинаются с 0). Если со строкой в **TStrings** или в **TStringList** связан объект, он остается связанным с той же строкой в новой позиции.

Пример

Приведенный ниже оператор перемещает первую строку списка **MyStrings** в конец списка.

```
MyStrings->Move(0, MyStrings->Count - 1);
```

MoveTo

Изменяет текущую позицию пера на заданную, ничего при этом не рисуя

Класс *TCanvas*

Прототип

```
void __fastcall MoveTo(int X, int Y);
```

Описание

Метод **MoveTo** изменяет текущую позицию пера **PenPos** на заданную точкой (X, Y). Это эквивалентно непосредственной установке свойства **PenPos**. При перемещении пера методом **MoveTo** ничего не рисуется.

MouseCapture

- См. раздел **TControl**.

Pie

Рисует заполненную замкнутую фигуру — сегмент окружности или эллипса

Класс *TCanvas*

Прототип

```
void __fastcall Pie(int X1, int Y1, int X2, int Y2,
                   int X3, int Y3, int X4, int Y4);
```

Описание

Метод **Pie** рисует замкнутую фигуру — сектор окружности или эллипса с помощью текущих параметров пера **Pen**. Фигура заполняется текущим значением **Brush**. Точки (X1, Y1) и (X2, Y2) определяют прямоугольник, описывающий эллипс. Начальная точка дуги определяется пересечением эллипса с прямой, проходящей через его центр и точку (X3, Y3). Конечная точка дуги определяется пересечением эллипса с прямой, проходящей через его центр и точку (X4, Y4). Дуга рисуется против часовой стрелки от начальной до конечной точки. Рисуются прямые, ограничивающие сегмент и проходящие через центр эллипса и точки (X3, Y3) и (X4, Y4).

В Windows 95/98 суммы $X1 + X2$, $Y1 + Y2$ и $X1 + X2 + Y1 + Y2$ не должны превышать 32768.

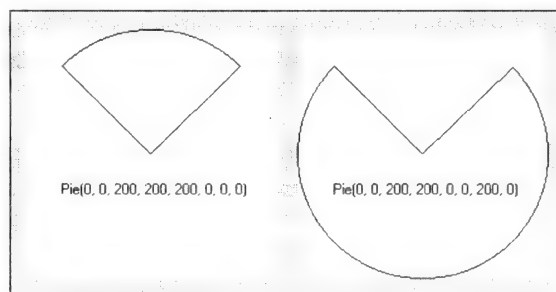
В Windows NT направление рисования дуги можно изменить на направление по часовой стрелке вызовом функции **SetArcDirection**.

Примеры

Операторы

```
Image1->Canvas->Pie(0, 0, 200, 200, 200, 0, 0, 0);
Image2->Canvas->Pie(0, 0, 200, 200, 0, 0, 200, 0);
```

дают результат, показанный на рисунке.



PolyBezier и PolyBezierTo

Рисуют на канве текущим пером кусочную кривую третьего порядка, сглаживающую заданное множество точек

Класс *TCanvas*

Прототип

```
void __fastcall PolyBezier(const Windows::TPoint * Points,
                           const int Points_Size);
void __fastcall PolyBezierTo(const Windows::TPoint * Points,
                             const int Points_Size);
```

Описание

Методы **PolyBezier** и **PolyBezierTo** складывают множество точек **Points_Size**, содержащихся в массиве **Points**, кусочной кривой третьего порядка. При этом функция **PolyBezier** точно отображает первую и последнюю точку, а **PolyBezierTo** — только последнюю. Число точек **Points_Size** для каждого метода должно быть строго определенным (хотя это, к сожалению, не указано в справке C++Builder): для **PolyBezier** оно должно быть кратно 3 (т.е. $i*3$), а для **PolyBezierTo** — на единицу меньше числа, кратного 3 (т.е. $i*3-1$). Если число точек не равно заданному, то функции просто ничего не рисуют.

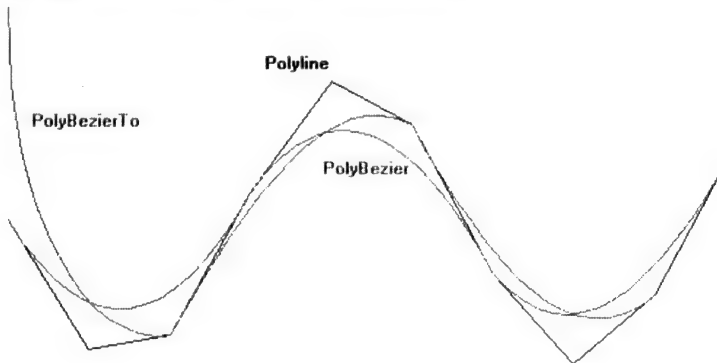
Исходя из этого при произвольном числе точек **N** имеет смысл автоматически приводить число точек к требуемому. Например, такими операторами:

```
PolyBezier(points, (N/3)*3);
PolyBezierTo(points, (N/3)*3-1);
```

В этих операторах число точек **N** за счет округления при целочисленном делении автоматически приводится к требуемому.

Пример

Ниже приведен рисунок аппроксимаций функции $-\sin(x)$ методами **PolyBezier**, **PolyBezierTo** и **Polyline** и код их построения.



```
const N = 10, Lx = 500, Ly = 100, T = 10;
TPoint points[N];

// заполнение массива
for(int i = 0; i <= N; i++)
    points[i] = Point((int)(i * Lx / (N-1)),
                      (int)(sin((double)i * T / (N-1))*Ly) +
                      Image1->ClientHeight / 2);

// рисование
Image1->Canvas->Pen->Color = clBlack;
Image1->Canvas->Polyline(points, N-1);
```



```

Image1->Canvas->Pen->Color = clRed;
Image1->Canvas->PolyBezier(points, (N/3)*3);

Image1->Canvas->Pen->Color = clGreen;
Image1->Canvas->PolyBezierTo(points, (N/3)*3-1);

```

Polygon

Рисует на канве текущим пером замкнутую фигуру (многоугольник) по заданному множеству угловых точек, замыкая первую и последнюю точки и закрашивая внутреннюю область фигуры текущей кистью

Класс *Tcanvas*

Прототип

```

void __fastcall Polygon(const Windows::TPoint * Points,
                        const int Points_Size);

```

Описание

Метод **Polygon** рисует на канве замкнутую фигуру (полигон, многоугольник) по множеству угловых точек, заданному массивом **Points**. Первая из указанных точек соединяется прямой с последней. Этим метод **Polygon** отличается от метода **Polyline**, который не замыкает конечные точки. Рисование проводится текущим пером **Pen**. Внутренняя область фигуры закрашивается текущей кистью **Brush**.

Метод позволяет рисовать фигуру по точкам, хранящимся в массиве элементов типа **TPoint**.

Пример

Операторы

```

TPoint points[5];
points[0] = Point(30,150);
points[1] = Point(40,130);
points[2] = Point(50,140);
points[3] = Point(60,130);
points[4] = Point(70,150);
Image1->Canvas->Polygon(points,4);

```

рисуют на канве формы многоугольник по точкам, хранящимся в массиве **points**.

Polyline

Рисует на канве текущим пером кусочно-линейную кривую по заданному множеству точек

Класс *TCanvas*

Прототип

```

void __fastcall Polyline(const Windows::TPoint * Points,
                        const int Points_Size);

```

Описание

Метод **Polyline** рисует на канве кусочно-линейную кривую по множеству точек, заданному массивом **Points**. Отличие метод **Polyline** от метода **Polygon** заключается в том, что метод **Polygon** замыкает конечные точки, а метод **Polyline** — нет. Рисование проводится текущим пером **Pen**. Метод не изменяет текущей позиции **PenPos** пера **Pen**.

Метод позволяет рисовать кусочно-линейный график функции, хранящийся в массиве элементов типа **TPoint**.

То, что делает метод **Polyline**, можно сделать и с помощью методов **MoveTo** и **LineTo**, подведя сначала перо к первой точке а затем последовательно выполняя **LineTo**. Различие будет заключаться в том, что метод **Polyline** не изменит текущую позицию пера, а методы **MoveTo** и **LineTo** изменят.

Пример

Операторы

```
TPoint points[5];
points[0] = Point(30,150);
points[1] = Point(40,130);
points[2] = Point(50,140);
points[3] = Point(60,130);
points[4] = Point(70,150);
Image1->Canvas->Polyline(points,4);
```

рисуют кусочно-линейную кривую по четырем точкам, заданным функциями **Point** (см. раздел 15.3.3 главы 15) в массиве **points**.

Realign

См. **DisableAlign**.

Rectangle

Рисует на канве текущим пером прямоугольник и закрашивает его текущей кистью

Класс *TCanvas*

Прототип

```
void __fastcall Rectangle(int X1, int Y1, int X2, int Y2);
```

Описание

Метод **Rectangle** рисует на канве текущим пером **Pen** прямоугольник, верхний левый угол которого имеет координаты (**X1**, **Y1**), а нижний правый — (**X2**, **Y2**). Прямоугольник закрашивается текущей кистью **Brush**.

Рисование прямоугольника без рамки можно осуществить методом **FillRect**. Прямоугольник со скругленными углами рисуется методом **RoundRect**. Прямоугольник без внутренней закрашки рисуется методом **FrameRect**.

Пример

```
Image1->Canvas->Rectangle(10,10,210,110);
```

Refresh

Перерисовывает изображение компонента на экране

Класс *TControl*

Определение

```
void __fastcall Refresh(void);
```

Описание

Метод **Refresh** приводит к немедленной перерисовке изображения на экране. **Refresh** вызывает метод **Repaint**. Методы **Refresh** и **Repaint** взаимозаменяемы.

Remove

Удаляет элемент с заданным значением из списка **TList**.

Класс *TList*

Прототип

```
int __fastcall Remove(void * Item);
```

Описание

Функция **Remove** удаляет указатель, равный заданному параметру **Item**, из списка **TList**. Функцию можно использовать вместо метода **Delete**, когда не известен индекс, соответствующий удаляемому указателю. Функция возвращает индекс, который имел данный указатель до его удаления. Индексы всех последующих указателей в списке уменьшаются на 1. Свойство **Count** также уменьшается на 1.

Если массив содержит несколько одинаковых указателей, то удаляется только первое вхождение этого указателя.

Repaint

Перерисовывает изображение компонента на экране

Класс *TControl*

Определение

```
virtual void __fastcall Repaint(void);
```

Описание

Вызов метода **Repaint** приводит к немедленной перерисовке изображения на экране. Если свойство **ControlStyle** компонента включает **csOpaque**, компонент перерисовывает себя сам. В противном случае **Repaint** вызывает метод **Invalidate**, а затем метод **Update**.

ReplaceDockedControl

Встраивает компонент на место другого уже встроенного

Класс *TControl*

Определение

```
enum TAlign {alNone, alTop, alBottom, alLeft, alRight, alClient};
```

```
bool __fastcall ReplaceDockedControl(TControl* Control,
                                     TwinControl* NewDockSite,
                                     TControl* DropControl,
                                     TAlign ControlSide);
```

Описание

Вызов метода **ReplaceDockedControl** приводит к встраиванию данного компонента на место другого уже встроенного, указанного параметром **Control**. При этом вытесненный компонент перемещается в другой контейнер **NewDockSite**. Параметр **DropControl** конкретизирует место встраивания вытесненного компонента в новом контейнере. Например, если **NewDockSite** — компонент **TPageControl**, то параметр **DropControl** должен указывать страницу размещения. **DropControl** может указываться равным **NULL**. Параметр **ControlSide** определяет выравнивание вытесненного компонента в **DropControl** или в **NewDockSite** (если **DropControl** = **NULL**). Это значение можно получить вызовом метода **GetDockEdge** контейнера.

Метод **ReplaceDockedControl** производит те же самые операции, что и метод **ManualDock**, примененный к **Control** с теми же параметрами **NewDockSite**, **DropControl** и **ControlSide**, плюс метод **ManualDock**, примененный к данному компоненту и размещающий его в позиции компонента **Control**. Но, метод **ReplaceDockedControl** компактнее, более эффективен и предотвращает неприятное мерцание при перестроении компонентов.

RoundRect

Рисует на канве прямоугольную рамку со скругленными углами

Класс *TCanvas*

Прототип

```
void __fastcall RoundRect(int X1, int Y1, int X2, int Y2,  
                          int X3, int Y3);
```

Описание

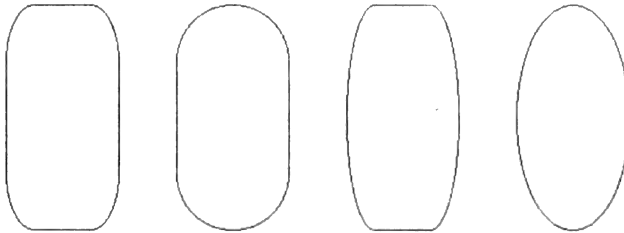
Метод **RoundRect** рисует на канве прямоугольную рамку со скругленными углами, используя текущие установки пера **Pen** и заполняя площадь фигуры текущей кистью **Brush**. Рамка определяется прямоугольником с координатами углов (**X1,Y1**) и (**X2,Y2**). Углы скругляются с помощью эллипсов с шириной **X3** и высотой **Y3**.

Если задать ширину эллипса **X3 ; X2 - X1**, то верхняя и нижняя границы рамки окажутся целиком скругленными (без прямолинейной части). Если **Y3 ; Y2 - Y1**, то же самое произойдет с левой и правой границами рамки. Если же оба измерения эллипса не меньше размеров рамки, то будет рисоваться просто эллипс. Но, конечно, для рисования эллипса лучше использовать метод **Ellipse**. Если один из размеров эллипса задать нулевым, то будет рисоваться прямоугольная рамка. Но, конечно, для такой рамки лучше использовать метод **Rectangle**.

Пример

Следующие операторы вызывают изображение, показанное на приведенном ниже рисунке:

```
Image1->Canvas->RoundRect(10,10,110,210,50,100);  
Image1->Canvas->RoundRect(160,10,260,210,100,100);  
Image1->Canvas->RoundRect(310,10,410,210,50,200);  
Image1->Canvas->RoundRect(460,10,560,210,100,200);
```



SaveToClipboardFormat

Создает копию изображения в формате Clipboard

Классы *TGraphic, TBitmap, TIcon, TMetafile, TPicture*

Прототип

```
virtual void __fastcall SaveToClipboardFormat(Word &AFormat,  
                                              int &AData, HPALETTE &APalette) = 0;
```

Описание

Метод **SaveToClipboardFormat** создает копию изображения в формате Clipboard. Формат, указатель на данные и палитру возвращаются как параметры **AFormat, AData** и **APalette**. Стандартно зарегистрированные форматы: **CF_BITMAP** для битовых карт и **CF_METAFILEPICT** для метафайлов. Формат для нового типа графического объекта предварительно должен быть зарегистрирован методом **RegisterClipboardFormat**.

После применения метода **SaveToClipboardFormat** надо передать объекту **Clipboard** полученные значения **AFormat** и **AData** методом **SetAsHandle**. При этом надо не забыть включить в приложение директиву

```
#include <vcl\Clipbrd.hpp>.
```

Впрочем, записать изображение в Clipboard можно и проще, воспользовавшись методом **Assign** объекта **Clipboard** для объектов типов **TGraphic**, **TBitmap**, **TIcon**, **TMetafile**.

Примеры

```
#include <vcl\Clipbrd.hpp>
...
Word MyFormat;
THandle AData;
HPALETTE APalette;
...
Image1->Picture->Bitmap->SaveToClipboardFormat(MyFormat,
                                                AData, APalette);
Clipboard()->SetAsHandle(MyFormat,AData);
```

Приведенные операторы записывают в буфер обмена изображение, хранящееся в свойстве **Picture->Bitmap** компонента **Image1**, вместе с палитрой и регистрируют формат **MyFormat**. В дальнейшем его можно использовать для чтения изображения из буфера:

```
if (Clipboard()->HasFormat(MyFormat))
{
    try
    {
        Image2->Picture->Bitmap->LoadFromClipboardFormat(CF_BITMAP,
                                                         Clipboard()->GetAsHandle(MyFormat), 0);
    }
    catch (...)
    {
        ShowMessage("Загрузка изображения невозможна");
    }
}
else
    ShowMessage("В буфере не изображение по формату MyFormat");
```

Впрочем, записать изображение в буфер обмена можно и не используя метод **SaveToClipboardFormat**, например, оператором:

```
Clipboard()->Assign(Image1->Picture->Bitmap);
```

или

```
Clipboard()->Assign(Image1->Picture->Graphic);
```

SaveToFile — метод графических объектов

Сохраняет графическое изображение в файле

Классы *TGraphic*, *TMetafile*, *TPicture*

Прототип

```
virtual void __fastcall SaveToFile(const AnsiString Filename);
```

Описание

Метод **SaveToFile** сохраняет изображение графического объекта в файле **Filename**.

Пример

Применение метода **SaveToFile** для типа **TPicture** позволяет, в частности, преобразовывать один тип графических файлов в другой. В качестве примера ниже

приведен код, создающий на основании файла пиктограммы **My.ico** файл точечного изображения **My.bmp**.

```
TIcon *ico = new TIcon();
try
{
    ico->LoadFromFile("My.ico");
    Image1->AutoSize = false;
    Image1->Width = ico->Width;
    Image1->Height = ico->Height;
    Image1->Canvas->Draw(0,0,ico);
    Image1->Picture->SaveToFile("My.bmp");
}
finally
{
    delete ico;
}
```

SaveToStream

Сохраняет графическое изображение в потоке

Классы *TGraphic*, *TBitmap*, *TIcon*, *TMetafile*, *TPicture*

Прототип

```
virtual void __fastcall SaveToStream(Classes::TStream* Stream) = 0;
```

Описание

Метод **SaveToStream** сохраняет в потоке **Stream** изображение графического объекта. Используется, например, для сохранения в объекте **TBlobStream** графического поля из набора данных.

ScaleBy

Масштабирует оконный элемент и все содержащиеся в нем компоненты

Класс *TWinControl*

Определение

```
void __fastcall ScaleBy(int M, int D);
```

Описание

Метод **ScaleBy** масштабирует оконный элемент и все содержащиеся в нем компоненты. Масштабируются такие свойства компонента, как **Width** и **Height**, определяющие его размер. Свойства **Top** и **Left** остаются неизменными. Масштабируется также размер шрифта, если только в компоненте не установлено **ParentFont = true**. В последнем случае шрифт наследуется от родительского компонента и поэтому не изменяется.

Если компонент является контейнером, содержащим другие компоненты, то эти дочерние компоненты также масштабируются. Причем у них изменяются не только **Width** и **Height**, но также пропорционально изменяются **Top** и **Left**, определяющие их местоположение. Если во всех дочерних компонентах установлено **ParentFont = true**, а в компоненте-контейнере **ParentFont = false**, то пропорционально изменяются и шрифты всех компонентов (но, конечно, не непрерывно, а скачками, доступными тому или иному типу шрифта).

Параметры **M** and **D** определяют соответственно множитель и делитель масштаба. Например, чтобы уменьшить размеры на 10% начального значения, можно задать **M** равным 9, а **D** равным 10 (9/10). Если же вы хотите увеличить размер на 1/3, то можно задать **M=133** и **D=100** (133/100) или **M=4** и **D=3** (4/3).

Подробнее метод рассмотрен в главе 4 в разделе 4.2.5.

Примеры

1. Оператор

```
Edit1->ScaleBy(11,10);
```

масштабирует окно редактирования **Edit1**. В любом случае при выполнении этого оператора увеличивается на 10% длина окна (свойство **Width**), что обеспечивает возможность наблюдать и редактировать в нем более длинный текст. Высота окна (свойство **Height**) будет изменяться пропорционально, если свойство компонента **AutoSize** равно **false**. В противном случае высота определяется только размером шрифта и при постоянном шрифте будет неизменной. А размер шрифта будет меняться, только если свойство компонента **ParentFont** равно **false**, т.к. иначе шрифт определяется родительским компонентом.

2. Приведенный ниже обработчик события **OnKeyUp** окна редактирования **Edit1** дает пользователю возможность менять длину окна. При нажатии комбинаций клавиш Alt-U и Alt-D пользователь увеличивает или уменьшает длину окна.

```
void __fastcall TForm1::Edit1KeyUp(TObject *Sender, WORD &Key,
                                   TShiftState Shift)
{
    if((Key == 'U') && (Shift.Contains(ssAlt)))
        Edit1->ScaleBy(11,10);
    else if((Key == 'D') && (Shift.Contains(ssAlt)))
        Edit1->ScaleBy(10,11);
}
```

Когда **Edit1** находится в фокусе, при нажатии пользователем клавиши Alt и клавиши U (в любом регистре и независимо от переключения на латинский или русский язык) длина окна редактирования увеличится на 10%, а при нажатии Alt и D соответственно уменьшится.

ScaleControls

Масштабирует дочерние компоненты оконного элемента, не изменяя масштаба самого элемента

Класс *TWinControl*

Определение

```
void __fastcall ScaleControls(int M, int D);
```

Описание

Метод **ScaleControls** масштабирует все компоненты, содержащиеся в оконном элементе, не изменяя масштаба самого элемента. Метод **ScaleControls** вызывает метод **ChangeScale** для каждого дочернего компонента. Отличается от метода **ScaleBy** только тем, что не изменяет масштаба самого элемента.

Параметры **M** and **D** определяют соответственно множитель и делитель масштаба. Например, чтобы уменьшить размеры на 10% начального значения, можно задать **M** равным 9, а **D** равным 10 (9/10). Если же вы хотите увеличить размер на 1/3, то можно задать **M**=133 и **D**=100 (133/100) или **M**=4 и **D**=3 (4/3).

Подробнее см. в разделах **ChangeScale** и **ScaleBy**.

ScreenToClient

Преобразует координаты экрана в координаты клиентской области компонента

Класс *TControl*

Определение

```
struct TPoint
```

```

{
    int x;
    int y;
};

Windows::TPoint __fastcall ScreenToClient(
    const Windows::TPoint &Point);

```

Описание

Метод **ScreenToClient** преобразует координаты точки в системе координат экрана (начало координат — левый верхний угол экрана) в систему координат клиентской области компонента (начало координат — левый верхний угол клиентской области).

Совместно с обратной функцией **ClientToScreen** метод может использоваться для пересчета координат точки экрана из системы координат клиентской области одного компонента в систему координат клиентской области другого компонента.

Пример

```
P = Comp2->ScreenToClient(Comp1->ClientToScreen(P));
```

Оператор пересчитывает координату точки экрана **P** из системы координат компонента **Comp1** в систему координат компонента **Comp2**.

ScrollBy

Сдвигает содержимое оконного элемента

Класс *TWinControl*

Определение

```
void __fastcall ScrollBy(int DeltaX, int DeltaY);
```

Описание

Метод **ScrollBy** сдвигает содержимое оконного элемента, включая все его дочерние компоненты. Метод применим ко всем оконным элементам, но чаще всего используется для наследников класса **TScrollingWinControl**.

Параметры **DeltaX** и **DeltaY** определяют величину сдвига по горизонтали и вертикали соответственно. Положительные значения параметров задают сдвиг вправо и вниз, отрицательные значения — влево и вверх.

Пример

Оператор

```
ScrollBy(1,1);
```

сдвигает все компоненты на форме на один пиксель вправо и на один вниз.

SelectFirst

Передает фокус дочернему компоненту, первому в последовательности табуляции

Класс *TWinControl*

Определение

```
void __fastcall SelectFirst(void);
```

Описание

Метод **SelectFirst** передает фокус дочернему компоненту, первому в последовательности табуляции. Он вызывает метод **FindNextControl**, передавая ему параметр, равный **NULL**. В результате метод **FindNextControl** возвращает первый компонент в последовательности табуляции, после чего этот компонент делается активным.

Пример
Оператор

```
SelectFirst();
```

активизирует первый в последовательности табуляции компонент на форме.

SelectNext

Передает фокус дочернему компоненту, следующему в последовательности табуляции за указанным

Класс *TWinControl*

Определение

```
void __fastcall SelectNext(TWinControl* CurControl,  
                           bool GoForward, bool CheckTabStop);
```

Описание

Метод **SelectNext** передает фокус дочернему компоненту, следующему в последовательности табуляции за тем, который указан параметром **CurControl**. Параметр **GoForward** определяет направление поиска: при значении **true** поиск ведется вперед, при значении **false** - назад.

Параметр **CheckTabStop** указывает, должен ли искомый компонент иметь свойство **TabStop**, равным **true**. Если значение **CheckTabStop** равно **true**, очередной компонент должен иметь значение **TabStop**, равное **true**, или поиск прекращается.

Если метод **SelectNext** не смог найти компонент в соответствии с заданными значениями **GoForward** и **CheckTabStop**, то фокус остается на компоненте **CurControl**.

SendCancelMode

Прерывает модальное состояние элемента управления

Класс *TControl*

Определение

```
void __fastcall SendCancelMode(TControl* Sender);
```

Описание

Вызов метода **SendCancelMode** прерывает модальное состояние элемента управления. Ряд элементов, реализованных в библиотеке визуальных компонентов C++Builder, поддерживают модальное состояние, при котором пользователь должен ответить элементу прежде, чем он сможет общаться с другими объектами формы. Метод **SendCancelMode** позволяет завершить модальное состояние без каких-либо действий со стороны пользователя.

SendToBack

Переносит компонент ниже других компонентов в Z-последовательности

Класс *TControl*

Прототип

```
void __fastcall SendToBack(void);
```

Описание

Метод **SendToBack** позволяет изменять последовательность перекрытия компонентов на форме и тем самым управлять видимостью компонентов.

Перекрывающиеся компоненты на форме размещаются поверх друг друга в так называемой Z-последовательности, соответствующей порядку размещения

компонентов в процессе проектирования. Например, если вы поместили в одно и то же место формы две кнопки одинаковых размеров, то видна будет только вторая из размещенных кнопок, поскольку она расположена в Z-последовательности выше. Применение во время выполнения приложения метода **SendToBack** к верхней кнопке переместит ее вниз в Z-последовательности и пользователю станет видна нижняя кнопка.

Если переносимый вниз компонент имел фокус, то он его потеряет при переносе.

Это справедливо по отношению к неоконным объектам, таким, как кнопки, метки, изображения и т.д., а также и к оконным компонентам, таким, как **Мемо**, **ComboBox** и др. Но все неоконные компоненты всегда расположены в Z-последовательности ниже оконных и метод **SendToBack** не может изменить это правило. Например, попытка перенести вниз методом **SendToBack** оконный компонент, под которым размещена метка, ни к чему не приведет.

Примеры

В разделе, посвященном методу **BringToFront**, также изменяющему последовательность компонентов, приведен ряд примеров. Во всех них вместо метода **BringToFront**, применяемому к нижнему компоненту, можно применять метод **SendToBack**, но к верхнему компоненту.

SetBounds

Устанавливает одновременно свойства **Left**, **Top**, **Width** и **Height**

Класс *TControl*

Определение

```
virtual void __fastcall SetBounds(int ALeft, int ATop,  
                                int AWidth, int AHeight);
```

Описание

Метод **SetBounds** изменяет одновременно все свойства компонента, определяющие его границу. Тот же эффект может быть достигнут совокупностью операторов изменения **Left**, **Top**, **Width** и **Height**. То, что метод **SetBounds** одновременно задает эти значения, не только позволяет получить более компактный код, но и дает возможность избежать перерисовки компонента после изменения каждого параметра в отдельности. При этом ликвидируется мерцание изображения, которое получается при поочередном изменении параметров.

Значения **Left**, **Top**, **Width** и **Height** задаются при вызове **SetBounds** как соответственно параметры **ALeft**, **ATop**, **AWidth** и **AHeight**.

Примеры, в которых требуется изменять сразу все эти свойства, см. в разделах **Visible** и **BringToFront**.

SetChildOrder

Изменяет позицию компонента в списке дочерних компонентов оконного элемента

Класс *TWinControl*

Определение

```
DYNAMIC void __fastcall SetChildOrder(  
                                Classes::TComponent* Child, int Order);
```

Описание

Метод **SetChildOrder** изменяет положение компонента, заданного параметром **Child**, в списке дочерних компонентов оконного элемента. Этот список — свойство **Controls** оконного элемента.

Параметр **Order** определяет индекс, присваиваемый компоненту **Child** (помните, что индексы начинаются с 0). Последовательность индексов остальных компонентов остается неизменной, но сами значения индексов «смыкаются», заполняя прежний индекс изменяемого компонента, и «раздвигаются» освобождая место для его нового индекса.

Пример

Если, например, компонент **Edit2** был в списке **Controls** формы вторым (т.е. имел индекс 1), то после выполнения оператора

```
SetChildOrder(Edit2, 3);
```

элемент с индексом 0 сохранит свой индекс, элементы с индексами 2 и 3 изменят индексы на 1 и 2, компонент **Edit2** будет иметь индекс 3, а индексы остальных элементов не изменятся.

SetFocus

Передает фокус элементу

Класс *TWinControl*

Определение

```
virtual void __fastcall SetFocus(void);
```

Описание

Метод **SetFocus** передает фокус данному компоненту, активизирует его.

Пример

Оператор

```
Edit1->SetFocus();
```

передает фокус компоненту **Edit1**.

SetZOrder

Перемещает компонент на вершину или в низ Z-последовательности

Классы *TControl*, *TWinControl*

Определение

```
DYNAMIC void __fastcall SetZOrder(bool TopMost);
```

Описание

Метод **SetZOrder** перемещает компонент на вершину или в низ Z-последовательности родительского элемента. Если родительского элемента нет, то элемент становится верхним или нижним окном экрана. Таким образом, этот метод перестраивает перекрывающиеся компоненты или окна.

Если параметр **TopMost** равен **true**, то элемент перемещается на вершину; в противном случае он перемещается вниз.

При этом надо иметь в виду, что оконные элементы всегда расположены в Z-последовательности выше не оконных. Так что при выполнении этого метода перемещения происходят только в этих допустимых пределах.

Show

Делает видимым невидимый компонент

Класс *TControl*

Определение

```
void __fastcall Show(void);
```

Описание

Метод **Show** делает видимым ранее невидимый компонент. Он задает значение **true** свойству **Visible** и проверяет, является ли видимым родительский компонент.

Примеры использования видимых и невидимых компонентов см. в разделе **Visible**.

StretchDraw

Рисует графическое изображение в указанную прямоугольную область канвы, подгоняя размер изображения под заданную область

Класс *TCanvas*

Прототип

```
void __fastcall StretchDraw(const Windows::TRect &Rect,  
                             TGraphic* Graphic);
```

Описание

Метод **StretchDraw** рисует на канве изображение, содержащееся в объекте, указанном параметром **Graphic**, в прямоугольную область, указанную параметром **Rect**. При этом размер изображения подгоняется под размер заданной области. Этим метод **StretchDraw** отличается от метода **Draw**, который оставляет размер неизменным.

Объект **Graphic** может быть типа битовой матрицы, пиктограммы или мета-файла. Если объект — битовая матрица типа **TBitmap**, то при переносе изображения учитывается режим копирования, установленный свойством канвы **CopyMode**.

Пример

Оператор

```
Image1->Canvas->StretchDraw(Rect(10,10,110,110),Bitmap1);
```

рисует на канве компонента **Image1** изображение из компонента **Bitmap1** в область с координатами углов (10, 10) и (110, 110). При этом размер изображения подгоняется под заданный размер области — квадрат со стороной 100.

TextExtent

Возвращает длину и высоту в пикселях текста, который предполагается написать на канве текущим шрифтом

Класс *TCanvas*

Прототип

```
struct TSize  
{  
    LONG cx;  
    LONG cy;  
};
```

```
TSize __fastcall TextExtent(const AnsiString Text);
```

Описание

Функция **TextExtent** возвращает структуру типа **TSize**, содержащую длину и высоту в пикселях текста **Text**, который предполагается написать на канве (см. **Canvas**) текущим шрифтом. Это позволяет перед выводом текста на канву определить размер надписи и расположить ее и другие элементы изображения наилучшим образом.

Только высоту или только длину текста можно определять соответственно методами **TextHeight** и **TextWidth**.

Пример

```
String st = Edit1->Text;
Canvas->TextOut((ClientWidth - Canvas->TextExtent(st).cx) / 2,
               Canvas->TextExtent(st).cy, st);
```

Эти операторы выводят на канву формы текст, набранный пользователем в окне редактирования **Edit1**, выравнивая его при любом шрифте по середине ширины канвы (формы) и отступив одну строчку сверху.

TextHeight

Возвращает высоту в пикселях текста, который предполагается написать на канве текущим шрифтом

Класс *TCanvas*

Прототип

```
int __fastcall TextHeight(const AnsiString Text);
```

Описание

Функция **TextHeight** возвращает высоту в пикселях текста **Text**, который предполагается написать на канве **Canvas** текущим шрифтом. Это позволяет перед выводом текста на канву определить размер надписи и расположить ее и другие элементы изображения наилучшим образом.

Имеется еще метод **TextExtent**, возвращающий одновременно и высоту, и длину текста. Метод **TextHeight** возвращает то же, что **TextExtent(Text).cy**.

Примеры

```
String st = Edit1->Text;
Image1->Canvas->TextOut(
    Image1->ClientWidth - Image1->Canvas->TextExtent(st).cx) / 2,
    Image1->Canvas->TextExtent(st).cy, st);
```

Эти операторы выводят текст, набранный пользователем в окне редактирования **Edit1**, в канву компонента **Image1**, выравнивая его при любом шрифте по середине ширины канвы и отступив одну строчку сверху.

См. также пример в описании метода **TextRect**.

TextOut

Пишет указанную строку текста на канве, начиная с указанной позиции

Класс *TCanvas*

Прототип

```
void __fastcall TextOut(int X, int Y, const AnsiString Text);
```

Описание

Функция **TextOut** пишет строку текста **Text** на канве (см. **Canvas**), начиная с позиции с координатами (**X**, **Y**). Надпись делается в соответствии с текущими установками шрифта **Font**. Фон надписи определяется установками текущей кисти **Brush**. Текущая позиция **PenPos** пера **Pen** перемещается к концу выведенного текста.

Для выравнивания позиции текста на канве можно использовать методы, дающие перед выводом высоту и длину текста в пикселях: методы **TextExtent**, **TextHeight** и **TextWidth**.

Если цвет кисти в момент вывода текста отличается от того, которым закрашена канва, то текст получится выведенным в цветной прямоугольной рамке. Но ее размеры будут точно равны размерам надписи. Если требуется более красивая рамка с отступом от текста или если надо ограничить выводимый текст размерами определенной рамки, следует применять метод **TextRect**.

Пример Оператор

```
Image1->Canvas->TextOut(10, 10, s);
```

выводит текст, хранящийся в строковой переменной **s**, на канву компонента **Image1**, начиная с позиции (10, 10).

См. также примеры в описаниях методов **TextExtent**, **TextHeight**, **TextWidth** и **TextRect**.

TextRect

Пишет указанную строку текста на канве, начиная с указанной позиции и усекая текст, выходящий за пределы указанной прямоугольной области

Класс *TCanvas*

Прототип

```
void __fastcall TextRect(const Windows::TRect &Rect,  
                        int X, int Y, const AnsiString Text);
```

Описание

Функция **TextRect** пишет строку текста **Text** на канве (см. **Canvas**), начиная с позиции с координатами (**X**, **Y**) — это левый верхний угол надписи. Часть текста, не помещающаяся в прямоугольную область **Rect**, усекается. Надпись делается в соответствии с текущими установками шрифта **Font**. Пространство внутри области **Rect** закрашивается текущей кистью **Brush**.

Для выравнивания позиции текста внутри области на канве можно использовать методы, дающие перед выводом высоту и длину текста в пикселях: методы **TextExtent**, **TextHeight** и **TextWidth**.

Примеры

1.

```
int X1,Y1,X2,Y2;  
String st = Edit1->Text;  
X1 = 100;  
Y1 = 100;  
X2 = 200;  
Y2 = 150;  
Image1->Canvas->Brush->Color = clRed;  
Image1->Canvas->TextRect (Rect(X1,Y1,X2,Y2),  
                        X1+(X2-X1-Image1->Canvas->TextWidth(st)) / 2,  
                        Y1+(Y2-Y1-Image1->Canvas->TextHeight(st)) / 2, st);
```

Приведенный код рисует в заданном месте канвы компонента **Image1** красный прямоугольник и внутри него в центре пишет методом **TextRect** текст, введенный пользователем в окно редактирования **Edit1**. Если текст оказывается длиннее ширины прямоугольника, то он усекается. В данном примере будет видна только середина текста, так как текст выровнен по центру прямоугольника.

2. Если в приведенном примере заменить оператор на

```
Image1->Canvas->TextRect (Rect(X1-5,Y1-5,  
                        X1+Image1->Canvas->TextWidth(st)+5,  
                        Y1+Image1->Canvas->TextHeight(st)+5),  
                        X1, Y1, st);
```

то текст будет выводиться полностью при любой его длине в красной прямоугольной области, на 5 пикселей отступающей во все стороны от текста. Именно этим отступом, делающим надпись более красивой, этот оператор отличается от более простого оператора

```
TextOut(X1,Y1,st);
```

использующего метод **TextOut**.

TextWidth

Возвращает длину в пикселях текста, который предполагается написать на канве текущим шрифтом

Класс *TCanvas*

Прототип

```
int __fastcall TextWidth(const AnsiString Text);
```

Описание

Функция **TextWidth** возвращает длину в пикселях текста **Text**, который предполагается написать на канве (см. **Canvas**) текущим шрифтом. Это позволяет перед выводом текста на канву определить размер надписи и расположить его и другие элементы изображения наилучшим образом.

Имеется еще метод **TextExtent**, возвращающий одновременно и высоту, и длину текста. Метод **TextWidth** возвращает то же, что **TextExtent(Text).cx**.

Примеры

```
String st = Edit1->Text;  
Canvas->TextOut(ClientWidth - Canvas->TextExtent(st).cx) / 2,  
                Canvas->TextExtent(st).cy, st);
```

Эти операторы выводят на канву формы текст, набранный пользователем в окне редактирования **Edit1**, выравнивая его при любом шрифте по середине ширины канвы и отступив одну строчку сверху.

См. также пример в описании метода **TextRect**.

TryLock

Блокирует канву, если она не была блокирована, не разрешая другим нитям многопоточного приложения рисовать на ней

Класс *TCanvas*

Прототип

```
bool __fastcall TryLock(void);
```

Описание

Функция **TryLock** блокирует данную канву, не разрешая другим нитям многопоточного приложения рисовать на ней. Канва остается блокированной до снятия блокады вызовом метода **Unlock**.

Если канва не была блокирована, то функция **TryLock** устанавливает свойство **LockCount** в 1 и возвращает **true**. Если канва уже была блокирована, то функция **TryLock** возвращает **false**, не увеличивая **LockCount**. В этом ее отличие от метода **Lock**, который допускает многократное вложенное блокирование.

Поскольку блокировка не дает другим нитям рисовать на канве, производительность работы приложения может за счет этого снизиться. Так что не надо злоупотреблять блокировками. Их следует применять только тогда, когда есть вероятность нежелательных наложений операций, выполняемых в разных нитях приложения с несколькими потоками.

Unlock

Уменьшает на единицу значение свойства **LockCount**, способствуя тем самым разблокированию канвы, когда **LockCount** станет равным 0

Класс *TCanvas*

Прототип

```
void __fastcall Unlock(void);
```

Описание

Метод **Unlock** уменьшает на единицу значение свойства **LockCount**, способствуя тем самым разблокированию канвы, заблокированной ранее методами **Lock** или **TryLock**. Блокировка канвы не разрешает другим нитям многопоточного приложения рисовать на ней. Канва остается заблокированной до тех пор, пока свойство **LockCount** не станет равным 0. А единственный способ уменьшения **LockCount** — вызов метода **Unlock**.

Если канва была заблокирована однократно, то вызов **Unlock** немедленно разблокирует ее. Если же были вложенные вызовы **Lock**, то канва будет разблокирована после стольких вызовов **Unlock**, сколько раз ранее вызывался **Lock**.

Update

Немедленно перерисовывает компонент

Класс *TWinControl*

Определение

```
virtual void __fastcall Update(void);
```

Описание

Метод **Update** вызывает немедленную перерисовку компонента, не ожидая завершения каких-то других процессов или прихода от Windows сообщений о перерисовке. Этим, в основном, этот метод и отличается от **Repaint**.

Для перерисовки вызывается функция **UpdateWindow** API Windows.

16.3 События

OnChange — событие класса TCanvas

Событие происходит сразу после изменения изображения на канве

Класс *TCanvas*

Определение

```
typedef void (__closure *TNotifyEvent)(System::TObject* Sender);  
__property Classes::TNotifyEvent OnChange
```

Описание

Событие **OnChange** наступает после изменения изображения на канве. При вызове любого метода рисования осуществляется следующая последовательность операций:

1. Наступает событие **OnChanging**.
2. Вызванный метод канвы **TCanvas** делает изменения в изображении.
3. Наступает событие **OnChange**.

Событие канвы **OnChange** наступает при изменении именно самого изображения, а не свойств канвы. Такие свойства канвы, как объекты **Font** — шрифт, **Brush** — кисть и **Pen** — перо имеют свои собственные события **OnChange**.

OnChange — событие класса TGraphicsObject

Событие наступает после изменения графического объекта

Классы *TGraphicsObject, TGraphic*

Определение

```
typedef void (__closure *TNotifyEvent)(System::TObject* Sender);  
__property Classes::TNotifyEvent OnChange
```


Описание

Обработчик события **OnChange** должен осуществить необходимые операции при изменении графического объекта и отразить его новые установки.

OnChangeing

См. событие канвы **OnChange**.

OnClick

Событие соответствует щелчку мыши на компоненте и некоторым другим действиям пользователя

Класс *TControl*

Определение

```
typedef void (__closure *TNotifyEvent) (System::TObject* Sender);
__property Classes::TNotifyEvent OnClick
```

Описание

Обычно событие **OnClick** наступает, если пользователь нажал и отпустил основную кнопку мыши, когда указатель мыши находился на компоненте. Это событие происходит также, если:

- Пользователь выбрал элемент в таблице, дереве, списке, выпадающем списке, нажав клавишу со стрелкой.
- Пользователь нажал клавишу пробела, когда кнопка или индикатор были в фокусе.
- Пользователь нажал клавишу Enter, а активная форма имеет кнопку по умолчанию, указанную свойством **Default**.
- Пользователь нажал клавишу Esc, а активная форма имеет кнопку прерывания, указанную свойством **Cancel**.
- Пользователь нажал клавиши быстрого доступа к кнопке или индикатору. Например, если свойство **Caption** индикатора записано как «&Полужирный» и символ **П** подчеркнут, то нажатие пользователем комбинации клавиш Alt-П вызовет событие **OnClick** в этом индикаторе.
- Приложение установило в **true** свойство **Checked** радиокнопки **RadioButton**.
- Приложение изменило свойство **Checked** индикатора **CheckBox**.
- Вызван метод **Click** элемента меню.

Для формы событие **OnClick** наступает, если пользователь щелкнул на пустом месте формы или на недоступном компоненте.

Параметр обработчика **Sender** содержит объект, в котором произошло событие, и может использоваться для дифференцированной реакции на события в разных компонентах.

Пример

Один обработчик события **OnClick** может использоваться для обработки событий в различных компонентах. Если при этом требуется различать, в каком компоненте произошло событие, можно использовать параметр **Sender**, как в приведенном чисто демонстрационном примере, отображающем сообщение о том, в каком компоненте произошло событие:

```
ShowMessage("OnClick в "+((TControl *)Sender)->Name);
```

В реальной программе, аналогичным образом проанализировав имя компонента, вы можете предусмотреть для разных компонентов разную реакцию.

OnCreate

Событие происходит при создании формы

Класс *TCustomForm*

Определение

```
typedef void (__closure *TNotifyEvent)(System::TObject* Sender);  
__property Classes::TNotifyEvent OnCreate
```

Описание

Событие **OnCreate** возникает в момент создания формы и может использоваться для выполнения каких-то процедур настройки ее самой или содержащихся на ней компонентов. Если в обработчике этого события создаются какие-то объекты, они должны разрушаться, освобождая память, в обработчике события **OnDestroy**.

Последовательность событий при создании формы, имеющей значение свойства **Visible**, равное **true**: **OnCreate**, **OnShow**, **OnActivate**, **OnPaint**.

OnDbClick

Событие соответствует двойному щелчку мыши на компоненте

Класс *TControl*

Определение

```
typedef void (__closure *TNotifyEvent)(System::TObject* Sender);  
__property Classes::TNotifyEvent OnDbClick
```

Описание

Событие **OnDbClick** наступает, если пользователь осуществил двойной щелчок: дважды с коротким интервалом нажал и отпустил основную кнопку мыши, когда указатель мыши находился на компоненте. К одному и тому же компоненту нельзя написать обработчики событий **OnClick** и **OnDbClick**, поскольку первый из них всегда перехватит первый из щелчков.

Параметр **Sender** содержит объект, в котором произошло событие, и может использоваться для дифференцированной реакции на события в разных компонентах (см. пример в разделе **OnClick**).

OnDragDrop

Событие наступает в момент отпускания перетаскиваемого компонента над данным компонентом

Класс *TControl*

Определение

```
typedef void (__closure *TDragDropEvent)  
(System::TObject* Sender,  
 System::TObject* Source,  
 int X, int Y);  
__property TDragDropEvent OnDragDrop
```

Описание

Событие **OnDragDrop** наступает в момент отпускания перетаскиваемого компонента над данным компонентом. В обработчике события надо описать, что в этот момент должно произойти. Параметр **Source** соответствует перетаскиваемому объекту, а параметр **Sender** — объекту, над которым объект был отпущен. Параметры **X** и **Y** содержат координаты позиции курсора мыши над компонентом в системе координат клиентской области этого компонента.

Примеры

1. Пусть на форме имеется несколько списков типа **TListBox** и вы хотите позволить пользователю перемещать строки из одного списка в другой. Это можно сделать следующим образом.

Во всех списках задаются значения свойств **DragMode**, равные **dmAutomatic**. Это обеспечивает автоматическое начало перетаскивания.

Далее для одного из списков пишется обработчик события **OnDragOver** вида:

```
void __fastcall TForm1::ListBox1DragOver(TObject *Sender,
                                         TObject *Source, int X, int Y, TDragState State,
                                         bool &Accept)
{
    Accept = Source->ClassNameIs("TListBox");
}
```

Этот обработчик указывает, что на данный компонент можно перетаскивать объекты типа **TListBox**.

Во всех остальных списках в событии **OnDragOver** указывается этот же обработчик.

Далее для одного из списков пишется обработчик события **OnDragDrop** вида:

```
void __fastcall TForm1::ListBox1DragDrop(TObject *Sender,
                                         TObject *Source, int X, int Y)
{
    TListBox *S = (TListBox *)Source;
    ((TListBox*)Sender)->Items->Add(
        S->Items->Strings[S->ItemIndex]);
    S->Items->Delete(S->ItemIndex);
}
```

Первые два оператора обработчика добавляют в список строку, выделенную в списке-источнике. Если пишется не универсальный обработчик, а предназначенный только для данного компонента, то первый оператор можно удалить, а во втором **((TListBox*)Sender)** и **S** заменить на имя компонента **ListBox1**. Третий оператор удаляет перенесенную строку из источника (если требуется не перенос, а только копирование строк из одного списка в другой, то этот оператор не нужен).

Во всех остальных списках в событии **OnDragDrop** указывается этот же обработчик.

После этого, заполнив списки, можно запускать приложение. Пользователь сможет перетаскивать строки между любыми имеющимися списками.

2. Если приведенный выше пример по каким-то соображениям желательно осуществлять не в автоматическом режиме, а в режиме ручного управления (например, перетаскивание возможно только в каком-то конкретном режиме работы приложения), то отличия в реализации примера заключаются в следующем.

Во всех списках задаются значения свойств **DragMode**, равные **dmManual**. Это обеспечивает управление началом перетаскивания.

Затем в одном из списков задается обработчик события **OnMouseDown** вида:

```
void __fastcall TForm1::ListBox3MouseDown(TObject *Sender,
                                           TMouseButton Button, TShiftState Shift, int X, int Y)
{
    if ((Button == mbLeft) && <проверка какого-нибудь условия>)
        ((TControl*)Sender)->BeginDrag(false, 5);
}
```

В этом обработчике первое условие проверяет, что нажата именно левая кнопка мыши, а в качестве второго можно задать какое-то условие, по которому нажатие кнопки мыши можно ассоциировать с началом перетаскивания (если вы предусматриваете подобное условие). Затем методом **BeginDrag** начинается перетаскивание. Поскольку в параметрах метода заданы значения **false** и **5**, то перетаскивание в действительности начнется только после сдвига мыши на 5 пикселей.

Во всех остальных списках в событии **OnMouseDown** указывается этот же обработчик.

Все остальные обработчики событий не отличаются от указанных в предыдущем примере.

3. Ряд примеров использования технологии перетаскивания **Drag&Drop** вы можете найти в главе 4 в разделе 4.4.1.

OnDragOver

Событие относится ко времени, в течение которого пользователь перемещает перетаскиваемый объект над компонентом

Класс *TControl*

Определение

```
typedef void (__closure *TDragOverEvent)(  
    System::TObject* Sender, System::TObject* Source,  
    int X, int Y, TDragState State, bool &Accept);  
__property TDragOverEvent OnDragOver
```

Описание

Событие **OnDragOver** начинается в момент, когда перетаскиваемый объект пересек границу данного компонента и оказался внутри его контура. Заканчивается событие, когда объект, покидая компонент, пересек его границу. Обработчик события **OnDragOver** используется для того, чтобы дать сигнал о готовности компонента принять перетаскиваемый объект в случае, если пользователь отпустит его над данным компонентом. Если компонент готов принять объект, в обработчике надо задать значение параметра **Accept**, равное **true**. Впрочем, это значение по умолчанию равно **true**, так что его можно не задавать. Вообще в предельном случае обработчик может быть пустым, что будет означать готовность компонента принять любой объект. Но даже пустой обработчик нужен, так как иначе сообщения о приеме компонента приложение не получит.

Во время перетаскивания над компонентом объекта, который может быть принят, форма курсора мыши может изменяться, сигнализируя пользователю о готовности компонента принять объект. Чтобы это было так, надо до момента события **OnDragOver** (а обычно — во время проектирования) задать соответствующее значение свойства компонента **DragCursor**.

Параметр **Source** определяет перетаскиваемый объект, параметр **Sender** — сам компонент, параметры **X** и **Y** — координаты точки экрана в пикселях. Параметр **State** типа **TDragState** определяет состояние перетаскиваемого объекта по отношению к другим объектам. Возможны следующие состояния:

Значение	Описание
dsDragEnter	Курсор мыши входит в пределы компонента.
dsDragMove	Курсор мыши перемещается в пределах компонента.
dsDragLeave	Курсор мыши выходит за пределы компонента.

Пример

```
void __fastcall TForm1::ListBox1DragOver(TObject *Sender,
                                         TObject *Source, int X, int Y, TDragState State,
                                         bool &Accept)
{
    Accept = (((TControl*)Sender)->Name == "ListBox1");
}
```

Этот обработчик события **OnDragOver** сигнализирует о том, что компонент готов принять перетаскиваемый объект, если это компонент **ListBox1**.

Развернутый пример обработки событий при перетаскивании, в частности, события **OnDragOver**, вы можете найти в разделе **OnDragDrop**.

OnEndDrag

Событие наступает в момент прерывания или окончания перетаскивания компонента

Класс TControl**Определение**

```
typedef void (__closure *TEndDragEvent)(
    System::TObject* Sender,
    System::TObject* Target, int X, int Y);
__property TEndDragEvent OnEndDrag
```

Описание

Событие **OnEndDrag** наступает при любом окончании процесса перетаскивания компонента — успешном (компонент перетащен в приемник) или безуспешном (компонент отпущен над формой или компонентом, не способным его принять). Событие наступает в перетаскиваемом компоненте.

Обработка этого события не требуется для осуществления процесса перетаскивания. Соответствующий обработчик может быть написан, если требуется какое-то действие или сообщение, подтверждающее результат перетаскивания, или какая-то реакция в перетаскивавшемся компоненте.

Параметр **Sender** — это сам объект перетаскивания. Параметр **Target** — это компонент-приемник, если объект был им принят, или **NULL**, если перетаскивание закончилось неудачей. Параметры **X** и **Y** — координаты экрана в пикселях.

Пример

```
void __fastcall TForm1::ListBox1EndDrag(TObject *Sender,
                                         TObject *Target, int X, int Y)
{
    if (Target == NULL)
        ShowMessage("Перенесение объекта " +
                    ((TControl*)Sender)->Name + " прервано");
    else
        ShowMessage(((TControl*)Sender)->Name + " перенесен в "
                    + ((TControl*)Target)->Name);
}
```

В этом примере просто отображается сообщение о результатах перетаскивания типа «Перенесение объекта **ListBox1** прервано» или «**ListBox1** перенесен в **ListBox2**». Но, конечно, аналогичным образом можно предусмотреть любые действия.

OnEnter

Событие наступает в момент получения элементом фокуса

Класс *TWinControl*

Определение

```
typedef void (__closure *TNotifyEvent)(System::TObject* Sender);  
__property Classes::TNotifyEvent OnEnter
```

Описание

Событие **OnEnter** наступает в момент получения элементом фокуса. Это событие не наступает при переключениях между формами или между приложениями.

При переключениях между элементами, расположенными в разных контейнерах, например, на разных панелях, событие **OnEnter** сначала наступает для контейнера, а потом для содержащегося в нем элемента. Это противоположно последовательности событий **OnExit**, которые при переключении на компонент другого контейнера наступают сначала для компонента, а потом для контейнера.

Пример

Пусть форма имеет кнопку ОК и групповую панель, включающую три радиокнопки. Пусть в начальный момент активна кнопка ОК. Когда пользователь щелкнет на одной из радиокнопок, в кнопке ОК наступит событие **OnExit**, затем наступит событие **OnEnter** групповой панели, и только затем наступит событие **OnEnter** той кнопки, на которой щелкнули. Если после этого пользователь щелкнет на кнопке ОК, то сначала наступит событие **OnExit** радиокнопки, затем событие **OnExit** групповой панели, а затем событие **OnEnter** кнопки ОК.

OnExit

Событие наступает в момент потери элементом фокуса

Класс *TWinControl*

Определение

```
typedef void (__closure *TNotifyEvent)(System::TObject* Sender);  
__property Classes::TNotifyEvent OnExit
```

Описание

Событие **OnExit** наступает в момент потери элементом фокуса, в момент его переключения на другой элемент. Это событие не наступает при переключениях между формами или между приложениями.

Значение свойства **ActiveControl** изменяется прежде, чем происходит событие **OnExit**.

При переключениях между элементами, расположенными в разных контейнерах, например, на разных панелях, событие **OnExit** сначала наступает для элемента, а потом для содержащего его контейнера. Это противоположно последовательности событий **OnEnter**, которые при переключении из другого контейнера на компонент данного контейнера наступают сначала для контейнера, а потом для компонента.

Пример, поясняющий последовательность событий при переключениях фокуса, приведен в разделе **OnEnter**.

OnKeyDown

Событие наступает при нажатии пользователем любой клавиши

Класс *TWinControl*

Определение

```
enum Classes__1 { ssShift, ssAlt, ssCtrl, ssLeft, ssRight,  
                  ssMiddle, ssDouble };  
typedef Set<Classes__1, ssShift, ssDouble> TShiftState;
```

```
typedef void (__closure *TKeyEvent)(System::TObject* Sender,
                                   Word &Key, Classes::TShiftState Shift);

__property TKeyEvent OnKeyDown
```

Описание

Событие **OnKeyDown** наступает, если компонент находится в фокусе, при нажатии пользователем любой клавиши, включая функциональные и вспомогательные, такие, как Shift, Alt и Ctrl.

В обработчик события передаются, кроме обычного параметра **Sender**, указывающего на компонент, в котором произошло событие, также параметры **Key** и **Shift**. Параметр **Key** определяет нажатую клавишу клавиатуры. Для не алфавитно-цифровых клавиш используется виртуальный код API Windows. Эти коды и способы проверки параметра **Key** приведены в главе 15 в разделе 15.1.1. Коды не различают символы в верхнем и нижнем регистрах и не различают символы кириллицы и латинские.

Параметр **Shift** является множеством, которое может быть пустым или включать следующие элементы:

Элемент	Значение
ssShift	Нажата клавиша Shift.
ssAlt	Нажата клавиша Alt.
ssCtrl	Нажата клавиша Ctrl.

Значения элементов **Shift**, соответствующие нажатиям кнопок мыши, в данном событии не используются.

Примеры

Пусть вы хотите распознать комбинацию клавиш Alt-X. Для этого вы можете написать оператор:

```
if((Key == 'X') && Shift.Contains(ssAlt)) ... ;
```

Реакцию на нажатие пользователем клавиши Enter можно оформить одним из следующих операторов:

```
if(Key == 13) ... ;
```

или

```
if(Key == 0x0D) ... ;
```

или

```
if(Key == VK_RETURN) ... ;
```

OnKeyPress

Событие наступает при нажатии пользователем клавиши символа

Класс TWinControl

Определение

```
typedef void (__closure *TKeyPressEvent)(
                                   System::TObject* Sender, char &Key);
__property TKeyPressEvent OnKeyPress
```

Описание

Событие **OnKeyPress** наступает, если компонент находится в фокусе, при нажатии пользователем клавиши символа. Параметр **Key** в обработчике этого события имеет тип **Char** и соответствует символу нажатой клавиши. При этом различа-

ются символы в верхнем и нижнем регистрах и символы кириллицы и латинские. Клавиши, не отражаемые в кодах ASCII (функциональные клавиши и такие, как Shift, Alt, Ctrl), не вызывают этого события. Поэтому нажатие таких комбинаций клавиш, как, например, Shift-A, генерирует только одно событие **OnKeyPress**, при котором параметр **Key** равен «А». Для того, чтобы распознавать клавиши, не соответствующие символам, или комбинации клавиш, надо использовать обработчики событий **OnKeyDown** и **OnKeyUp**.

Следует отметить, что событие **OnKeyPress** заведомо наступает, если нажимается только клавиша символа или клавиша символа при нажатой клавише Shift. Если же клавиша символа нажимается одновременно с какой-то из вспомогательных клавиш, то событие **OnKeyPress** может не наступить (произойдут только события **OnKeyDown** при нажатии и **OnKeyUp** при отпускании) или, если и наступит, то укажет на неверный символ. Например, при нажатой клавише Alt событие **OnKeyPress** при нажатии символьной клавиши не наступает. А при нажатой клавише Ctrl событие **OnKeyPress** при нажатии символьной клавиши наступает, но символ не распознается.

Поскольку параметр **Key** передается в обработчик по ссылке, его можно изменять, передавая для дальнейшей стандартной обработки другой символ, как в приведенных ниже примерах.

Примеры

1. Пусть вы хотите, чтобы пользователь не мог вводить в окно редактирования **Edit1** какие-либо символы, кроме цифр. Это можно сделать, написав для **Edit1** следующий обработчик события **OnKeyPress**:

```
Set <char, '0', '9'> Dig;
Dig << '0' << '1' << '2' << '3' << '4'
    << '5' << '6' << '7' << '8' << '9';
if ( ! Dig.Contains(Key))
    Key = 0;
```

Этот оператор трансформирует все не цифровые символы в нулевые и они не будут отражаться в окне редактирования.

2. Приведенный ниже оператор обработчика события **OnKeyPress** переводит латинские символы в верхний регистр, независимо от того, в каком регистре набрал их пользователь:

```
Key = UpCase (Key);
```

Этот оператор действует только на латинские символы. Приведенный ниже аналогичный оператор действует и на латинский символы, и на символы кириллицы (подробнее см. в раздела 15.4.2.3 главы 15):

```
Key = AnsiUpperCase (Key) [1];
```

OnKeyUp

Событие наступает при отпускании пользователем любой клавиши

Класс *TWinControl*

Определение

```
enum Classes__1 { ssShift, ssAlt, ssCtrl, ssLeft, ssRight,
                  ssMiddle, ssDouble };
typedef Set<Classes__1, ssShift, ssDouble> TShiftState;

typedef void (__closure *TKeyEvent)(System::TObject* Sender,
                                     Word &Key, Classes::TShiftState Shift);

__property TKeyEvent OnKeyUp
```


Описание

Событие **OnKeyUp** наступает, если компонент находится в фокусе, при отпус­кании пользователем любой ранее нажатой клавиши, включая функциональные и вспомогательные, такие, как Shift, Alt и Ctrl.

В обработчик события передаются, кроме обычного параметра **Sender**, указы­вающего на компонент, в котором произошло событие, также параметры **Key** и **Shift**. Параметр **Key** определяет клавишу клавиатуры, которая отпускается. Для не алфавитно-цифровых клавиш используются виртуальные коды API Windows. Эти коды и способы проверки параметра **Key** приведены в главе 15 в раз­деле 15.1.1. Коды не различают символы в верхнем и нижнем регистрах и не разли­чают символы кириллицы и латинские.

Параметр **Shift** является множеством, которое может быть пустым или вклю­чать следующие элементы:

Элемент	Значение
ssShift	Отпускается клавиша Shift.
ssAlt	Отпускается клавиша Alt.
ssCtrl	Отпускается клавиша Ctrl.

Значения элементов **Shift**, соответствующих нажатиям кнопок мыши, в дан­ном событии не используются.

Событие **OnKeyUp** наиболее удобно, чтобы распознавать нажатые клавиши, особенно, комбинации клавиш. Надо только не забывать, что параметр **Key** имеет тип **word**, а не **char**, так что для распознавания надо использовать виртуальные коды. К тому же, надо учитывать, что виртуальный код не различает символы в верхнем и нижнем регистре и не реагирует на то, русский или английский язык включен в данный момент.

Примеры

Пусть вы хотите написать обработчик, который бы реагировал на нажатие клавиш Shift-Y. Проверить нажатые клавиши можно оператором:

```
if((Key == 'Y') && Shift.Contains(ssShift)) ... ;
```

Но он будет реагировать и на «Y», и на «y», и даже на русские буквы «Н» и «н», которые обычно расположены на той же клавише, что и латинская «Y». Для распознавания действительного символа надо использовать событие **OnKeyPress**.

OnMouseDown и OnMouseUp

Событие наступает в момент нажатия пользователем клавиши мыши над ком­понентом

Класс *TControl*

Определение

```
enum TMouseButton { mbLeft, mbRight, mbMiddle } ;

enum Classes__1 { ssShift, ssAlt, ssCtrl, ssLeft, ssRight,
                  ssMiddle, ssDouble } ;
typedef Set<Classes__1, ssShift, ssDouble> TShiftState;

typedef void ( __closure *TMouseEvent )(System::TObject* Sender,
                                       TMouseButton Button, Classes::TShiftState Shift, int X, int Y);

__property TMouseEvent OnMouseDown
```

Имеется также парное к данному событие **OnMouseUp**, наступающее при отпуске нажатой кнопки мыши над объектом. Его определение использует тот же тип **TMouseEvent**:

```
__property TMouseEvent OnMouseUp
```

Описание

Обработка событий **OnMouseDown** и **OnMouseUp** используется для операций, требуемых при нажатии и отпуске пользователем какой-нибудь кнопки мыши.

Если требуется различная обработка событий в зависимости от того, какая кнопка мыши нажата или какая нажата вспомогательная клавиша, можно анализировать параметры **Button** и **Shift**. Значения параметра **Button** определяют, какая кнопка мыши нажата: **mbLeft** — левая, **mbRight** — правая, **mbMiddle** — средняя. Параметр **Shift** представляет собой множество, содержащее помимо обозначения нажатой кнопки еще и обозначения нажатых одновременно с этим вспомогательных клавиш **Shift**, **Alt**, **Ctrl** (соответствуют элементам множества **ssShift**, **ssAlt**, **ssCtrl**), а также **ssDouble** — двойной щелчок. Параметры **X** и **Y** определяют координаты указателя мыши в клиентской области компонента. Параметр **Sender** — указатель на компонент, в котором произошло событие.

Примеры

1. Обработчик события **OnMouseDown** может использоваться для начала процесса перетаскивания компонента, если вы решили задать какое-то дополнительное условие (например, проверка каких-то опций), по которому можно начинать перетаскивание. В этом случае в компоненте вы задаете свойство **DragMode**, равным **dmManual**, что обеспечивает управление началом перетаскивания. Обработчик события **OnMouseDown** может иметь вид:

```
void __fastcall TForm1::ListBox3MouseDown(TObject *Sender,
                                           TMouseButton Button, TShiftState Shift, int X, int Y)
{
    if ((Button == mbLeft) && <проверка какого-нибудь условия>)
        ((TControl*) Sender)->BeginDrag(false, 5);
}
```

В приведенной структуре **if** первое условие (**Button == mbLeft**) можно заменить эквивалентным ему условием, проверяющим параметр **Shift**:

```
if (Shift.Contains(ssLeft)) ...;
```

2. Если вы пишете обработчик, описанный выше, но хотите, чтобы перетаскивание начиналось только в случае, когда пользователь нажал левую кнопку мыши при нажатой клавише **Alt**, оператор обработки может иметь вид:

```
if ((Button == mbLeft) && (Shift.Contains(ssAlt)))
    ((TControl*) Sender)->BeginDrag(false, 5);
```

OnMouseMove

Событие наступает при перемещении курсора мыши над компонентом

Класс *TControl*

Определение

```
enum Classes__1 { ssShift, ssAlt, ssCtrl, ssLeft, ssRight,
                  ssMiddle, ssDouble };
typedef Set<Classes__1, ssShift, ssDouble> TShiftState;

typedef void (__closure *TMouseMoveEvent)(System::TObject* Sender,
                                           Classes::TShiftState Shift, int X, int Y);
__property TMouseMoveEvent OnMouseMove
```

Описание

Обработчик события **OnMouseMove** пишется, если надо произвести какие-то операции при перемещении курсора мыши над компонентом.

Параметр **Shift**, являющийся множеством, содержит элементы, позволяющие определить, какие кнопки мыши и какие вспомогательные клавиши (Shift, Ctrl и Alt) нажаты в этот момент. Параметры **X** и **Y** определяют координаты указателя мыши в клиентской области компонента. Параметр **Sender** (источник события) — сам компонент.

Событие **OnMouseMove** возникает независимо от того, нажаты ли какие-то кнопки или клавиши. Правда, хотя это и не документировано в C++Builder, при нажатой правой кнопке мыши (а в некоторых типах мыши — при нажатой левой кнопке) это событие, почему-то, не наблюдается.

Пример

```
if (Shift.Contains(ssAlt))...
```

Этот оператор проверяет, не нажата ли клавиша Alt во время перемещения курсора мыши над компонентом, и, если нажата, то предпринимаются какие-то действия.

OnMouseUp

См. раздел **OnMouseDown** и **OnMouseUp**.

OnPaint

Событие наступает при получении сообщения Windows о необходимости перерисовать испорченное изображение

Классы *TCustomForm*, *TPaintBox*

Определение

```
typedef void (__closure *TNotifyEvent)(System::TObject* Sender);
```

```
__property Classes::TNotifyEvent OnPaint
```

Описание

Событие **OnPaint** наступает, когда приходит сообщение Windows о необходимости перерисовать испорченное изображение. Изображение может испортиться из-за временного перекрытия данного окна другим окном того же или стороннего приложения. Обработчик данного события должен перерисовать изображение. При перерисовке изображения канвы **Canvas** можно использовать ее свойство **ClipRect**, которое указывает область канвы, внутри которой изображение испорчено.

Примеры

Если копия изображения, отображаемого на канве, хранится в компоненте **Bitmap**, то обработчик события **OnPaint** для формы может иметь вид:

```
Canvas->Draw(0,0,Bitmap);
```

а для компонента **PaintBox**

```
PaintBox1->Canvas->Draw(0,0,Bitmap);
```

Более быстрая перерисовка получается при использовании свойства **ClipRect** канвы, например:

```
Canvas->CopyRect(Canvas->ClipRect,Bitmap->Canvas, Canvas->ClipRect);
```

OnProgress

События происходят при медленных процессах изменения графического изображения и позволяют построить индикатор хода процесса

Классы *TGraphic*, *TImage*, *TPicture*

Определение

```
enum TProgressStage {psStarting, psRunning, psEnding};

typedef void (__closure *TProgressEvent)(System::TObject* Sender,
                                         TProgressStage Stage,
                                         Byte PercentDone,
                                         bool RedrawNow,
                                         const Windows::TRect &R,
                                         const AnsiString Msg);

__property TProgressEvent OnProgress
```

Описание

События **OnProgress** наступают во время медленных процессов изменения графического изображения таких, как загрузка, сохранение, трансформация. Эти события позволяют построить в приложении индикатор хода процесса, обеспечивающий обратную связь с пользователем. Разработчики новых компонентов могут генерировать события **OnProgress**, вызывая защищенный метод **Progress**.

Параметр **Stage** указывает стадию процесса: начало, продолжение, окончание. Он может соответственно принимать значения **psStarting**, **psRunning** и **psEnding**. Если приложение предусматривает индикацию процесса, то можно создавать индикатор при **Stage = psStarting**, изменять его показания, пока **Stage = psRunning** и закрывать при **Stage = psEnding**. Параметр **PercentDone** показывает, какая примерно часть процесса выполнена. Этот параметр может использоваться в индикаторе процесса.

Параметр **RedrawNow** указывает, может ли изображение в данный момент быть успешно отображено на экране. Параметр **R** указывает область изображения, которая изменена и нуждается в перерисовке.

Параметр **Msg** содержит краткую справку о протекающем процессе. Например, **Loading**, **Storing** или **Reducing colors**. Строка **Msg** может быть и пустой.

OnStartDrag

Событие наступает, когда пользователь начинает процесс перетаскивания компонента

Класс *TControl*

Определение

```
typedef void (__closure *TStartDragEvent)(System::TObject* Sender,
                                           TDragObject* &DragObject);

__property TStartDragEvent OnStartDrag
```

Описание

Событие **OnStartDrag** наступает, когда пользователь нажал левую кнопку мыши над объектом и, не отпуская ее, начал смещать курсор мыши, т.е. начал перетаскивание.

Обработчик события **OnStartDrag** позволяет описать какие-то специальные действия, необходимые при начале перетаскивания. Параметр **Sender** является тем компонентом, который должен перетаскиваться или который содержит объект будущего перетаскивания.

Переменная **DragObject** по умолчанию равняется **NULL**. Это означает, что переноситься будет сам компонент или его объект. При этом автоматически создается объект типа **TDragControlObject**, с которым **C++Builder** осуществляет весь процесс перетаскивания. Но обработчик может создавать и новый объект перетаскивания, в котором определен какой-нибудь особый вид курсора и т.п.

16.4 Некоторые базовые классы и типы

Иерархия некоторых базовых классов библиотеки компонентов

Полную иерархию классов библиотеки компонентов VCL C++Builder привести в данной книге невозможно, поскольку она слишком обширна. Ниже приводится только небольшой ее фрагмент, содержащий наиболее интересные для пользователя базовые классы. Знание хотя бы общей характеристики этих классов помогает получать доступ к свойствам и методам объектов неизвестного класса, что нередко требуется и будет рассмотрено в следующем разделе.

TObject	Базовый класс всех объектов VCL. Не имеет никаких свойств, но имеет ряд полезных методов, применимых ко всем объектам: ClassName , ClassNameIs , ClassParent , Free , InheritsFrom
Exception	Базовый класс всех исключений. Определены свойства HelpContext и Message , а также конструкторы, используемые во всех исключениях
TPersistent	Базовый класс всех объектов VCL C++Builder, допускающих операцию присваивания или участвующих своими свойствами в операциях с потоками. Определен метод присваивания Assign
TComponent	Базовый класс всех компонентов. Определено свойство Name — имя компонента и Owner — владелец компонента
TControl	Базовый класс всех визуальных компонентов. Определено множество свойств, описывающих размещение компонента, его размеры, выравнивание, видимость, перетаскивание и т.д. Определено много свойств и все основные события, связанные с клавиатурой и мышью
TWinControl	Базовый класс всех оконных компонентов. Определены свойства и методы, характеризующие вид окна, фокусировку, последовательность табуляции, свойства окна как контейнера других компонентов, процессы перетаскивания
TCustomEdit	Базовый класс однострочных и многострочных окон редактирования. Определены свойства и методы редактирования текста
TCustomComboBox	Базовый класс выпадающих списков. Определены свойства и методы работы с элементами списков
TButtonControl	Базовый класс кнопок и индикаторов. Все свойства и методы наследуются от базовых компонентов

TGraphicControl	Базовый класс многих неоконных компонентов, включая метки, графические компоненты, быстрые кнопки. В нем определено свойство Canvas — холст, являющееся основой получения изображений
TMenu	Базовый класс главных и всплывающих меню. Определены свойства и методы доступа к разделам меню
TMenuItem	Базовый класс разделов меню. Определены их свойства и методы
TCommonDialog	Базовый класс диалогов. Определен общий для всех диалогов метод их выполнения — Execute
TField	Базовый класс полей наборов данных. Определены общие для всех полей свойства доступа: AsFloat , AsInteger и т.п.
TDataSet	Базовый класс наборов данных (TTable , TQuery , TStoredProc и др.). Определены общие для всех наборов данных свойства и методы
TStrings	Класс объектов, представляющих собой списки строк и используемых во многих компонентах C++Builder для различных свойств. Определены свойства и методы доступа и манипулирования строками

AnsiString — тип строк

В C++Builder тип строк **AnsiString** реализован как класс, объявленный в файле **vcl/dstring.h** и аналогичный типу длинных строк в Delphi. Это строки с нулевым символом в конце. При объявлении переменные типа **AnsiString** инициализируются пустыми строками.

Для **AnsiString** определены операции отношения **==**, **!=**, **>**, **<**, **>=**, **<=**. Сравнение производится с учетом регистра. Сравниваются коды символов, начиная с первого, и если очередные символы не одинаковы, строка, содержащая символ с меньшим кодом считается меньше. Если все символы совпали, но одна строка длиннее и в ней имеются еще символы, то она считается больше, чем более короткая.

Для **AnsiString** определены операции присваивания **=**, **+=** и операция склеивания строк (конкатенации) **+**. Определена также операция индексации **[]**. Индексы начинаются с 1. Например, если **S1** = «Привет», то **S1[1]** вернет 'П', **S1[2]** вернет 'р' и т.д.

Работа с классом **AnsiString** рассмотрена в главе 13 в разделе 13.4.2. Основные методы класса **AnsiString** (в описаниях методов через **S1** обозначена строка, метод которой используется):

Метод	Синтаксис / Описание	
AnsiCompare	int __fastcall AnsiCompare(const AnsiString& rhs) const Сравнивает данную строку S1 с rhs с учетом регистра. Сравнение зависит от текущих установок Windows и может отличаться от сравнения, осуществляемого операциями сравнения. Возвращает значение > 0 при S1 $>$ rhs , значение < 0 при S1 $<$ rhs и значение 0 при S1 = rhs	
AnsiCompare-IC	int __fastcall AnsiCompareIC(const AnsiString& rhs) const Осуществляет сравнение, аналогичное AnsiCompare , но без учета регистра	
AnsiLastChar	char* __fastcall AnsiLastChar() const Возвращает указатель на последний значащий символ. Поддерживает многобайтные символы	
AnsiPos	int __fastcall AnsiPos(const AnsiString& subStr) const Возвращает индекс первого символа первого вхождения subStr в S1 . Индексы начинаются с 1. Если subStr не содержится в S1 , возвращается 0. В отличие от Pos поддерживает многобайтные символы	
AnsiString	__fastcall AnsiString(аргумент) Конструктор класса. В зависимости от типа аргумента создает:	
	Аргумент	Создает
	Отсутствует	Пустую строку
	const char* src	Строку с нулевым символом в конце из массива символов
	const AnsiString& src	Копию AnsiString src
	const char* src, unsigned char len	Строку с нулевым символом в конце, являющуюся копией первых len символов из src
	const wchar_t* src	Строку с нулевым символом в конце из массива src символов типа wchar_t
	int src	Строку с нулевым символом в конце из массива src целых значений символов
	double src	Строку с нулевым символом в конце из массива src значений символов с плавающей запятой; преобразуются первые 15 значащих разрядов
c_str	char* __fastcall c_str()const Возвращает указатель на строку с нулевым символом в конце, содержащую те же символы, что в AnsiString	
CurrToStr	static AnsiString __fastcall CurrToStr(Currency value) Преобразует значение value типа Currency в строку	

Метод	Синтаксис / Описание
CurrToStrF	static AnsiString __fastcall CurrToStrF(Currency value, TstringFloatFormat format, int digits) Преобразует значение value типа Currency в строку, используя указанный формат преобразования чисел с плавающей запятой (см. в главе 14 раздел тип TstringFloatFormat). Параметр определяет задаваемое число разрядов. Функция соответствует функции CurrToStrF с заданной точностью 19 разрядов
Delete	void __fastcall Delete(int index, int count) Удаляет из строки, начиная с позиции index число символов, равное count
FloatToStrF	static AnsiString __fastcall FloatToStrF(long double value, TstringFloatFormat format, int precision, int digits) Преобразует значение value с плавающей запятой в строку, используя указанный формат (см. в главе 14 раздел тип TstringFloatFormat). Параметры precision и digits задают точность и число разрядов. Точность должна задаваться не более 7 для типа float , не более 15 для double и не более 18 для Extended . Число разрядов зависит от выбранного формата
Format	static AnsiString __fastcall Format(const AnsiString& format, const TVarRec *args, int size) Формирует строку, используя строку формата format и массив аргументов args
FormatFloat	static AnsiString __fastcall FormatFloat(const AnsiString& format, const long double& value) Преобразует значение value с плавающей запятой в строку, используя указанный формат format (см. в главе 14 раздел «Строка форматирования чисел с плавающей запятой»)
Insert	void __fastcall Insert(const AnsiString& str, int index) Вставляет в строку подстроку str , начиная с индекса index
IntToHex	static AnsiString __fastcall IntToHex(int value, int digits) Преобразует значение value в строку, содержащую минимум digits шестнадцатеричных цифр
IsDelimiter	bool __fastcall IsDelimiter(const AnsiString& delimiters, int index) const Возвращает true , если символ с индексом index является одним из разделителей, указанных в строке delimiters . Работает и для многобайтных символов
IsEmpty	bool __fastcall IsEmpty() const Возвращает true , если строка пустая
LastDelimiter	int __fastcall LastDelimiter(const AnsiString& delimiters) const Возвращает последний из символов строки, входящих в строку разделителей delimiters . Например, если <pre>AnsiString s = "c:\\filename.ext"; to s.LastDelimiter("\\.:");</pre> вернет 12 (индекс символа точки)

Метод	Синтаксис / Описание
Length	int __fastcall Length() const Возвращает число символов в строке
LowerCase	AnsiString __fastcall LowerCase() const Возвращает строку, в которой все символы приведены к нижнему регистру. Не влияет на исходную строку
Pos	int __fastcall Pos(const AnsiString& subStr) const Возвращает индекс первого символа первого вхождения subStr в S1 . Индексы начинаются с 1. Если subStr не содержится в S1 , возвращается 0. В отличие от AnsiPos не поддерживает многобайтные символы
SetLength	void __fastcall SetLength(int newLength) Усекает строку до newLength символов. Если исходная строка короче, то она не увеличивается
StringOfChar	static AnsiString __fastcall StringOfChar(char ch, int count) Возвращает строку, в которой символ ch повторен count раз. Например, <pre>AnsiString s = AnsiString::StringOfChar('A', 10);</pre> задаст строке s значение «AAAAAAAAAA»
SubString	AnsiString __fastcall SubString(int index, int count) const Возвращает подстроку, начинающуюся с символа в позиции index и содержащую count символов
ToDouble	double __fastcall ToDouble() const Преобразует строку в число с плавающей запятой. Если строка не соответствует формату числа с плавающей запятой, генерируется исключение EConvertError
ToInt	int __fastcall ToInt() const Преобразует строку в целое число. Если строка не соответствует формату целого числа, генерируется исключение EConvertError
ToIntDef	int __fastcall ToIntDef(int defaultValue) const Преобразует строку в целое число. Если строка не соответствует формату целого числа, возвращается значение по умолчанию defaultValue
Trim	AnsiString __fastcall Trim() const Возвращает строку, соответствующую исходной, но без пробельных символов до и после значащих символов
TrimLeft	AnsiString __fastcall TrimLeft() const Возвращает строку, соответствующую исходной, но без начальных пробельных символов
TrimRight	AnsiString __fastcall TrimRight() const Возвращает строку, соответствующую исходной, но без заключительных пробельных символов

Метод	Синтаксис / Описание
Unique	void __fastcall Unique() Делает строку уникальной, т.е. устанавливает число ссылок на нее (refcnt) в 1. Таким образом, на нее ссылается только один объект
UpperCase	AnsiString __fastcall UpperCase() const Возвращает строку, в которой все символы приведены к верхнему регистру. Не влияет на исходную строку
WideChar	wchar_t* __fastcall WideChar(wchar_t* dest, int destSize) const Преобразует строку в массив символов dest типа wchar_t и возвращает указатель на этот массив
WideCharBufSize	int __fastcall WideCharBufSize() const Возвращает размер буфера, требуемого для функции WideChar

Set — шаблон класса

Является шаблоном, реализующим встроенный класс Delphi, используемый в библиотеке компонентов VCL

Модуль *vcl/sysset.h*

Определение

```
template<class T, unsigned char minEl, unsigned char maxEl>
```

```
class __declspec(delphireturn) Set;
```

Описание

В шаблоне должно быть задано три параметра:

type	тип элементов множества (обычно int , char или enum)
minval	минимальное значение элемента множества (не менее 0)
maxval	максимальное значение элемента множества (не более 255)

Тип каждого объекта класса **Set** определяется всеми тремя параметрами. Если какие-то из этих параметров различаются, считается, что это объекты разных типов и их нельзя, например, сравнивать друг с другом. Например, если объявлены объекты

```
Set <char, 'A', 'C'> s1;  
Set <char, 'X', 'Z'> s2;
```

то оператор

```
if(s1 == s2) ...
```

будет воспринят как ошибка, поскольку типы **s1** и **s2** различны.

Для объектов класса **Set** определены методы, соответствующие всем операциям, допустимым для множеств (см. раздел 13.6 главы 13).

TBitmap — класс

Инкапсулирует битовую матрицу Windows (HBITMAP), включая палитру (HPALETTE)

Иерархия *TObject* — *TPersistent* — *TGraphic*

Модуль *graphics*

Описание

Класс **TBitmap** инкапсулирует битовую матрицу Windows, включая палитру. Обеспечивает быстрое и простое для пользователя выполнение операций создания, копирования, преобразования и сохранения битовой матрицы.

Свойства

Ниже приведен полный список свойств, определенных или переопределенных в **TBitmap**.

Свойство	Тип	Описание
Canvas	TCanvas	Определяет пространство (холст) для изображения битовой матрицы. Свойство только для чтения.
Empty	bool	Указывает, содержит ли объект битовую матрицу. Свойство только для чтения.
Handle	HBITMAP	Обеспечивает доступ к обработке битовых матриц в GDI Windows. Используется при вызовах функций API Windows.
HandleType	enum TBitmapHandleType {bmDIB, bmDDB}	Указывает, является ли битовая матрица DDB (Device Dependent Bitmap — аппаратно зависимой), или DIB (Device Independent Bitmap — аппаратно независимой). Может изменяться пользователем.
Height	int	Указывает высоту изображения в пикселях. Может изменяться пользователем, что вызывает создание копии матрицы с указанным размером.
IgnorePalette	bool	Определяет, использует ли матрица палитру. При установке в true ухудшается качество, но ускоряется рисование.
Modified	bool	Определяет, было ли модифицировано изображение после его загрузки.
MaskHandle	HBITMAP	Обеспечивает доступ к обработке битовых матриц в GDI Windows. Используется при вызовах функций API Windows. Свойство только для чтения.

Свойство	Тип	Описание
Monochrome	bool	Определяет, является ли битовая матрица монохромной (значение true).
Palette	HPALETTE	Управляет цветами битовой матрицы с 256 цветами. Если изображение не нуждается в палитре или не имеет палитры, то Palette = 0.
PixelFormat	enum TPixelFormat {pfDevice, pf1bit, pf4bit, pf8bit, pf15bit, pf16bit, pf24bit, pf32bit, pfCustom}	Определяет битовый формат отображения изображения. Используется для задания формата видеодрайверам, не способным прочитать собственный формат битовой матрицы.
ScanLine	void *	Обеспечивает доступ к отдельным строкам пикселей для их низкоуровневой обработки для матриц DIBs (Device Independent Bitmap — аппаратно независимых). Свойство только для чтения.
Transparent	bool	Определяет, должно ли изображение быть «прозрачным»
<u>TransparentColor</u>	TColor	Определяет, какой из цветов будет прозрачным при рисовании битовой матрицы. Читаемое значение зависит от значения TransparentMode .
<u>TransparentMode</u>	enum TTransparentMode {tmAuto, tmFixed}	Определяет, будет ли свойство <u>TransparentColor</u> сохраняться вместе с битовой матрицей.
Width	int	Указывает ширину изображения в пикселях. Может изменяться пользователем, что вызывает создание копии матрицы с указанным размером.

Методы

Ниже приведены основные методы, объявленные или переопределенные в классе **TBitmap**.

Метод	Описание
<u>Assign</u>	Копирует изображение из другого графического объекта, в частности, из буфера обмена Clipboard.
<u>Dormant</u>	Создает изображение битовой матрицы в памяти, чтобы освободить дескриптор матрицы и сэкономить ресурсы
<u>FreeImage</u>	Освобождает память, занятую кэшированием изображения, экономит ресурсы, но может вести к потере глубины цвета. См. описание и пример в разделе <u>Dormant</u> .

Метод	Описание
<u>Assign</u>	Копирует изображение из другого графического объекта, в частности, из буфера обмена Clipboard.
<u>LoadFromClipboardFormat</u>	Читает изображение из буфера обмена Clipboard в заданном формате.
<u>LoadFromFile</u>	Читает изображение из файла.
<u>LoadFromResourceID</u>	Загружает битовую карту из ресурса по указанному идентификатору.
<u>LoadFromResourceName</u>	Загружает битовую карту из ресурса по указанному имени.
<u>LoadFromStream</u>	Читает графическое изображение из указанного потока.
<u>Mask</u>	Преобразует изображение в монохромную маску.
<u>ReleaseHandle</u>	Возвращает дескриптор типа HBitmap и очищает объект TBitmap от этого дескриптора.
<u>ReleaseMaskHandle</u>	Возвращает дескриптор маски типа HBitmap и очищает объект TBitmap от этого дескриптора.
<u>ReleasePalette</u>	Возвращает дескриптор палитры типа HPalette и разрывает связь палитры с объектом TBitmap .
<u>SaveToClipboardFormat</u>	Сохраняет изображение в буфере обмена Clipboard в заданном формате.
<u>SaveToFile</u>	Сохраняет изображение в файле.
<u>SaveToStream</u>	Записывает изображение в поток.

События

Событие	Описание
<u>OnChange</u>	Событие при изменении графического объекта
<u>OnProgress</u>	События происходят при медленных процессах изменения графического изображения и позволяют построить индикатор хода процесса

TBrush — тип

Определяет свойства кисти: цвет и стиль заполнения фона окна

Иерархия *TObject* — *TPersistent* — *TGraphicsObject*

Модуль *graphics*

Описание

Тип **TBrush** инкапсулирует структуру **HBRUSH** Winodws и используется для заполнения форм заданным цветом и стилем. **TBrush** используется во многих объектах, в частности, в свойстве канвы кисть — **Brush**.

Свойства

Свойство	Тип	Описание
<u>Bitmap</u>	<u>TBitmap</u>	Указатель на внешнюю матрицу побитового отображения, используемую как шаблон заполнения.

Свойство	Тип	Описание
<u>Color</u>	<u>TColor</u>	Цвет кисти. По умолчанию — <code>clWhite</code> .
<u>Handle</u>	HBRUSH	Дескриптор кисти окна, определяющий доступ к дескриптору объекта GDI Windows.
<u>Style</u>	TBrushStyle	Определяет стиль заполнения окна.

Методы

В классе **TBrush** не введено каких-то принципиально новых методов. Переопределены такие общие методы, как Assign, конструктор и деструктор. Остальные методы наследуются от классов-предков.

События

Класс **TBrush** наследует от класса **TGraphicsObject** событие **OnChange**, наступающее после изменения графического объекта. Обработывая его графический объект должен учесть новые установки **TBrush**.

Пример

Следующие операторы изменяют цвет и стиль заполнения объекта **Image1->Canvas** — канвы компонента **Image1**:

```
Image1->Canvas->Brush->Style = bsCross;
Image1->Canvas->Brush->Color = clRed;
Image1->Canvas->FillRect(Rect(0, 0, Image1->Width, Image1->Height));
```

Последний из приведенных операторов заполняет методом FillRect всю поверхность канвы.

TCanvas — класс

Обеспечивает пространство (холст, канву) для создания, хранения и модификации графических объектов

Иерархия *TObject — TPersistent*

Модуль *graphics*

Описание

Класс **TCanvas** является основой графической подсистемы C++Builder. Канва обеспечивает:

- Загрузку и хранение графических изображений
- Создание новых и изменение хранимых изображений с помощью пера, кисти, шрифта
- Рисование и закраску различных фигур, линий, текстов
- Комбинирование различных изображений

Класс **TCanvas** имеет два дочерних класса — **TControlCanvas** и **TMetafileCanvas**, которые помогают в прорисовке управляющих элементов и в создании для объекта метафайла.

Свойства

Ниже приведен полный список свойств, определенных или переопределенных в **TCanvas**.

Свойство	Тип	Описание
<u>Brush</u>	<u>TBrush</u>	Определяет цвет и стиль заполнения замкнутых фигур и фона

Свойство	Тип	Описание
CanvasOrientation	enum TCanvasOrientation {coLeftToRight, coRightToLeft}	Определяет обычную (слева направо) и восточную (справа налево) ориентацию канвы и ее координат. Свойство только для чтения
ClipRect	TRect	Определяет доступную область рисования на канве и область, подлежащую перерисовке при событии OnPaint . Свойство только для чтения
CopyMode	int	Определяет режим копирования графического изображения на канву
Font	TFont	Определяет атрибуты шрифта, которым выводится текст
LockCount	int	Определяет, сколько раз блокирована канва в многопоточных приложениях. Свойство только для чтения
Pen	TPen	Определяет свойства пера, рисующего линии и фигуры
PenPos	TPoint	Определяет текущую позицию пера
Pixels	TColor	Определяет цвета пикселей
TextFlags	int	Определяет способ вывода текста на канву

Методы

Ниже приведены основные методы, объявленные в классе **TCanvas**.

Метод	Описание
Arc	Рисует дугу окружности или эллипса
BrushCopy	Копирует часть изображения битовой матрицы на данную канву, заменяя указанный цвет в изображении на значение, установленное для кисти канвы
Chord	Рисует замкнутую фигуру, ограниченную дугой окружности или эллипса и хордой
CopyRect	Копирует часть изображения с другой канвы на данную
Draw	Рисует графическое изображение в указанную позицию канвы
DrawFocusRect	Рисует изображение прямоугольника в виде, используемом для отображения рамки фокуса, операцией xor
Ellipse	Рисует окружность или эллипс
FillRect	Заполняет указанный прямоугольник канвы, используя текущее значение кисти Brush
FloodFill	Закрашивает текущей кистью замкнутую область канвы, определенную цветом
FrameRect	Рисует на канве текущей кистью прямоугольную рамку
LineTo	Рисует на канве прямую линию, начинающуюся с текущей позиции пера и кончающуюся указанной точкой

Метод	Описание
<u>Lock</u>	Блокирует канву, не разрешая другим нитям многопоточного приложения рисовать на ней
<u>MoveTo</u>	Изменяет текущую позицию пера на заданную, ничего не рисуя
<u>Pie</u>	Рисует сектор окружности или эллипса
<u>PolyBezier</u>	Сглаживают множество точек кусочной кривой третьего порядка, сохраняя первую и последнюю точку
<u>PolyBezierTo</u>	Сглаживают множество точек кусочной кривой третьего порядка, сохраняя последнюю точку
<u>Polygon</u>	Рисует замкнутую фигуру с кусочно-линейной границей
<u>Polyline</u>	Рисует кусочно-линейную кривую
<u>Rectangle</u>	Рисует прямоугольник
<u>RoundRect</u>	Рисует прямоугольник со скругленными углами
<u>StretchDraw</u>	Рисует графическое изображение в указанную прямоугольную область канвы, подгоняя размер изображения под заданную область
<u>TextExtent</u>	Возвращает длину и высоту в пикселях текста, который предполагается написать на канве текущим шрифтом
<u>TextHeight</u>	Возвращает высоту в пикселях текста, который предполагается написать на канве текущим шрифтом
<u>TextOut</u>	Пишет указанную строку текста на канве, начиная с указанной позиции
<u>TextRect</u>	Пишет указанную строку текста на канве, начиная с указанной позиции и усекая текст, выходящий за пределы указанной прямоугольной области
<u>TextWidth</u>	Возвращает длину в пикселях текста, который предполагается написать на канве текущим шрифтом
<u>TryLock</u>	Блокирует канву, если она не была блокирована, не разрешая другим нитям многопоточного приложения рисовать на ней
<u>Unlock</u>	Уменьшает на единицу значение свойства LockCount , способствуя тем самым разблокированию канвы, когда LockCount станет равным 0

События

Событие	Описание
<u>OnChange</u>	Событие после изменения изображения
<u>OnChanging</u>	Событие перед изменением изображения

TCColor — тип

Определяет цвет объекта

Модуль *Graphics*

Определение

```
enum TColor {clMin=-0x7fffffff-1, clMax=0x7fffffff};
```

Описание

Тип **TColor** используется для описания цвета объекта. Применяется в свойствах **Color** многих компонентов, в подсвойствах объекта типа **TFont**, при прорисовке изображений, в таблицах и т.д.

В модуле *Graphics* определено множество констант типа **TColor**. Одни из них непосредственно определяют цвета (например **clBlue** — синий), другие определяют цвета элементов окон, которые могут меняться в зависимости от выбранной пользователем палитры цветов Windows (например, **clBtnFace** — цвет поверхности кнопок). Полный перечень этих констант с пояснениями см. в разделе **Color**.

Вместо использования этих констант можно задавать **TColor** как 4-байтовое шестнадцатеричное число, три младших разряда которого представляют собой интенсивности синего, зеленого и красного цвета соответственно. Например, значение **0x00FF0000** соответствует чистому синему цвету, **0x0000FF00** — чистому зеленому, **0x000000FF** — чистому красному. **0x00000000** — черный цвет, **0x00FFFFFF** — белый.

Если старший байт равен нулю (00), то берется ближайший к заданному цвет из системной палитры. Если старший байт равен единице (01), то берется ближайший к заданному цвет из текущей палитры. Если старший байт равен двум (02), то берется ближайший к заданному цвет из логической палитры контекста данного устройства.

TComponent — базовый класс компонентов

Базовый класс всех компонентов C++Builder

Иерархия *TObject — TPersistent*

Модуль *classes*

Описание

Класс **TComponent** является прародителем всех компонентов C++Builder. Он инкапсулирует наиболее общие свойства и методы компонентов, включая:

- Возможность включать компонент в палитру компонентов и работать с ним при визуальном проектировании.
- Способность быть владельцем других компонентов или управляться другими компонентами.
- Возможности обмена с потоками и файлами.
- Возможность служить оболочкой элементов ActiveX и других объектов.

Объекты типа **TComponent** не создаются. Класс **TComponent** используется как базовый, когда объявляется класс невидимого компонента, который может присутствовать в палитре компонентов и применяться в процессе проектирования. Для создания визуальных компонентов в качестве базового используется класс **TControl** или его потомки. Для создания оконных компонентов в качестве базового используется класс **TWinControl** или его потомки.

Свойства

Свойство	Тип	Описание
ComObject	_di_IUnknown	Защищенное свойство, возвращающее ссылку на интерфейс в компонентах, поддерживающих стандарт COM

Свойство	Тип	Описание
<u>ComponentCount</u>	int	Число компонентов, которыми владеет данный компонент. Равно количеству элементов в массиве Components . На 1 меньше индекса последнего компонента ComponentIndex , поскольку индексы отсчитываются от 0. Может использоваться вместе с ComponentIndex в циклах, когда надо изменить какие-то свойства всех компонентов
<u>ComponentIndex</u>	int	Индекс компонента в массиве Components владельца данного компонента. Отсчитывается от 0, т.е. индекс первого компонента — 0. Может использоваться вместе с ComponentCount в циклах, когда надо изменить какие-то свойства всех компонентов
<u>Components</u>	TComponent *	Массив компонентов, которыми владеет данный компонент. Позволяет сослаться на любой компонент с помощью его ComponentIndex
ComponentState	TComponentState	Состояние компонента в процессе визуального проектирования. Свойство только для чтения. Во время выполнения не используется
ComponentStyle	TComponentStyle	Определяет множество флагов стиля компонента, в частности, стиль csInheritable указывает, что компонент может принадлежать форме
DesignInfo	int	Используется только в среде C++Builder при проектировании форм. В приложениях не используется
<u>Name</u>	AnsiString	Имя компонента, по которому производится ссылка на него из других компонентов
Owner	TComponent *	Определяет владельца данного компонента. Форма является владельцем всех расположенных на ней компонентов. В свою очередь Application является владельцем всех форм. Когда освобождается память, занимавшаяся владельцем, автоматически освобождается память всех компонентов, которыми он владел. Свойство только для чтения
<u>Tag</u>	int	Это свойство не используется в C++Builder. Разработчик может использовать его по своему усмотрению
VCLComObject	void *	Используется только в среде C++Builder компонентов, поддерживающих стандарт COM

Методы

В классе **TComponent** определено множество методов. Ниже приводятся те из них, которые определены или переопределены в **TComponent**, причем только те, которые наиболее часто используются пользователями при работе с компонентами. Остальные используются для внутренних потребностей C++Builder.

Метод	Описание
DestroyComponents	Удаляет из памяти все компоненты, которыми владеет данный компонент. Непосредственно этот метод не используется — он автоматически вызывается, когда вы удаляете компонент ключевым словом delete
FindComponent	Ищет в списке Components компонент с заданным именем
FreeNotification	Гарантирует, что указанный в вызове компонент будет разрушен. Используется только по отношению к компонентам, расположенным на других формах. Для компонентов на текущей форме вызывается автоматически
InsertComponent	Добавляет указанный компонент в конец списка компонента-владельца. При визуальном проектировании вызывается автоматически. Специально может потребоваться вызов этого метода только при добавлении компонента в список другого владельца
RemoveComponent	Удаляет указанный компонент из списка компонента-владельца. При визуальном проектировании вызывается автоматически. Специально может потребоваться вызов этого метода только при удалении компонента из списка другого владельца

Класс **TComponent** наследует также метод **Assign** класса **TPersistent** и другие методы своих предшественников.

TControl — базовый класс визуальных компонентов

Абстрактный базовый класс всех визуальных компонентов C++Builder

Иерархия *TObject — TPersistent — TComponent*

Модуль *controls*

Описание

Класс **TControl** является базовым классом для всех визуальных компонентов C++Builder, т.е. для компонентов, которые пользователь может видеть и которыми манипулирует во время выполнения приложения. Все они имеют общие свойства, методы и события, определяющие место их размещения, расцветку, реакцию на нажатие клавиш или кнопок мыши и т.д.

Защищенные свойства и методы класса **TControl** используются в их потомках. Если требуется создать новый класс визуального компонента, его надо создавать как производный от **TControl** или от его потомков.

Свойства

Класс **TControl** имеет следующие основные свойства:

Свойство	Тип	Описание
<u>Action</u>	TBasicAction *	Определяет действие, связанное с данным управляющим элементом
<u>Align</u>	enum TAlign { alNone, alTop, alBottom, alLeft, alRight, alClient }	Определяет способ выравнивания компонента в контейнере (родительском компоненте)
<u>Anchors</u>	enum TAnchors { akLeft, akTop, akRight, akBottom }	Определяет привязку данного компонента к родительскому при изменении размеров последнего
<u>AutoSize</u>	bool	Определяет, будет ли высота элемента автоматически адаптироваться к размеру символов текста
<u>BoundsRect</u>	TRect	Определяет координаты углов компонента в координатах содержащего его контейнера
<u>Caption</u>	System::AnsiString	Строка текста, идентифицирующая компонент для пользователя. Обычно это надпись на метке, кнопке и др. компонентах
<u>ClientHeight</u>	int	Высота клиентской области в пикселях
<u>ClientOrigin</u>	Windows::TPoint	Координаты положения на экране левого верхнего угла клиентской области компонента. Свойство только для чтения
<u>ClientRect</u>	Windows::TRect	Определяет координаты углов клиентской области компонента
<u>ClientWidth</u>	int	Горизонтальный размер клиентской области в пикселях
<u>Color</u>	Graphics::TColor	Цвет фона компонента
<u>Constraints</u>	TSizeConstraints	Позволяет задавать ограничения на допустимые изменения размеров компонента при изменениях размеров окна приложения
<u>ControlState</u>	TControlState	Характеризует текущее состояние компонента во время выполнения приложения. Используется при создании новых классов
<u>ControlStyle</u>	TControlStyle	Определяет характеристики стиля компонента. Используется при создании новых классов
<u>Cursor</u>	TCursor	Определяет вид курсора мыши, при попадании его в область компонента
<u>DesktopFont</u>	bool	Определяет, использует ли компонент для отображения текста изображение шрифта Windows

Свойство	Тип	Описание
<u>DockOrientation</u>	enum TDockOrientation { doNoOrient, doHorizontal, doVertical }	Определяет позицию данного встраиваемого компонента относительно других встроенных компонентов. Компоненты могут встраиваться горизонтально или вертикально
<u>DragCursor</u>	TCursor	Определяет вид курсора мыши, при попадании его в область компонента в процессе перетаскивания
<u>DragKind</u>	TDragKind = (dkDrag, dkDock)	Определяет, будет ли объект перетаскиваться по технологии Drag&Drop, или Drag&Doc
<u>DragMode</u>	TDragMode = (dmManual, dmAutomatic)	Определяет автоматическое или программное начало процесса перетаскивания
<u>Enabled</u>	bool	Определяет, реагирует ли компонент на события, связанные с мышью, клавиатурой и таймером
<u>Floating</u>	bool	Определяет, находится ли компонент в состоянии «плавающего» окна (см. главу 4 раздел 4.4.2). Свойство только для чтения
<u>FloatingDockSiteClass</u>	TMetaClass*	Определяет класс временного компонента, управляющего «плавающим» окном
<u>Font</u>	TFont	Определяет атрибуты шрифта
<u>Height</u>	int	Высота компонента в пикселях
<u>Hint</u>	AnsiString	Определяет текст подсказки
<u>HostDockSite</u>	TWinControl*	Определяет контейнер, в который встроен данный компонент. Задание HostDockSite приводит к немедленному встраиванию компонента в указанный контейнер. Если компонент никуда не встроен, HostDockSite = NULL
<u>IsControl</u>	bool	Определяет, сохраняет ли форма свои специфические свойства в поток. Свойство защищенное. Используется при создании новых компонентов
<u>Left</u>	int	Координата левого края компонента в пикселях
<u>LRDockWidth</u>	int	Ширина компонента, когда он в последний раз размещался в контейнере горизонтально. Свойство только для чтения

Свойство	Тип	Описание
MouseCapture	bool	Определяет, может ли компонент захватываться мышью. Свойство защищенное. Используется при создании новых компонентов
Name	AnsiString	Имя компонента
Parent	TWinControl	Определяет родительский компонент, в площади которого располагается данный компонент
ParentColor	bool	Определяет, что для компонента будет заимствован цвет родительского компонента. См. разделы Color и Parent
ParentFont	bool	Включает и выключает использование шрифта родительского компонента
ParentShowHint	bool	Включает и выключает использование свойства ShowHint родительского компонента
PopupMenu	TPopupMenu	Определяет связанный с компонентом объект всплывающего меню
ScalingFlags	enum Controls_9 { sfLeft , sfTop , sfWidth , sfHeight , sfFont }	Показывает, какие атрибуты компонента должны масштабироваться. Используется при разработке новых компонентов
ShowHint	bool	Разрешает или запрещает показывать окно подсказки
TBDockHeight	Integer	Высота компонента, когда он в последний раз размещался в контейнере вертикально. Свойство только для чтения
Text	TCaption = string	Текст, связанный с данным компонентом
Top	int	Координата верхнего края компонента в пикселях
UndockHeight	int	Высота компонента, которая была в последний раз, когда он отображался плавающим окном. Свойство только для чтения
UndockWidth	int	Ширина компонента, которая была в последний раз, когда он отображался плавающим окном. Свойство только для чтения
Visible	bool	Делает компонент видимым или невидимым
Width	int	Горизонтальный размер компонента в пикселях

Свойство	Тип	Описание
WindowProc	TWndMethod	Содержит оконную процедуру обработки сообщений, поступающих компоненту. Используется при создании новых компонентов
<u>WindowText</u>	PChar	Содержит текст, связанный с данным компонентом

Помимо перечисленных свойств класс **TControl** наследует также ряд свойств **TComponent**, из которых можно отметить **ComponentCount**, **ComponentIndex**, **Components**, **Owner**, **Tag** и другие.

Методы

Ниже приводится таблица только тех методов, которые могут применяться пользователями компонентов или разработчиками не очень сложных компонентов. Помимо перечисленных в классе определено еще много методов, интересных только для разработчиков сложных новых компонентов.

Метод	Описание
<u>BeginDrag</u>	Начинает процесс перетаскивания компонента
<u>BringToFront</u>	Переносит компонент в Z-последовательности выше других компонентов на той же форме
Changed	Используется, чтобы послать сообщение CM_CHANGED родительскому компоненту, если в свойствах данного компонента сделаны какие-то изменения, на которые должен прореагировать родительский компонент
<u>ChangeScale</u>	Изменяет масштаб компонента
Click	Вызывает обработчик события OnClick при щелчке мыши. Используется при проектировании новых классов
<u>ClientToScreen</u>	Преобразует координаты клиентской области в координаты экрана
DbClick	Вызывает обработчик события OnDbClick при двойном щелчке мыши. Используется при проектировании новых классов
DoEndDock	Вызывает обработчик события OnEndDock . Используется при проектировании новых классов
DoEndDrag	Вызывает обработчик события OnEndDrag . Используется при проектировании новых классов
DoStartDock	Вызывает обработчик события OnStartDock . Используется при проектировании новых классов
DoStartDrag	Вызывает обработчик события OnStartDrag . Используется при проектировании новых классов
DragCanceled	Прерывает перетаскивание. Используется при проектировании новых классов
DragDrop	Вызывает обработчик события OnDragDrop . Используется при проектировании новых классов

Метод	Описание
Dragging	Определяет, перетаскивается ли компонент в данный момент
EndDrag	Завершает (успешно или неуспешно) перетаскивание. Используется при проектировании новых классов
GetControlsAlignment	Возвращает вариант выравнивания текста в компоненте
GetDockEdge	Возвращает характер выравнивания рамок встраиваемых компонентов, перемещаемых над данным контейнером. Вызывается автоматически
GetTextBuf	Записывает в заданный буфер фиксированного размера значение свойства Text . Используется, если нужна обратная совместимость с 16-битными кодами
GetTextLen	Возвращает длину строки свойства Text , необходимую для задания размера буфера в методе GetTextBuf
Hide	Делает компонент невидимым
Invalidate	Вызывает полную перерисовку испорченного изображения компонента
MouseDown	Вызывает обработчик события OnMouseDown . Используется при проектировании новых классов
MouseMove	Вызывает обработчик события OnMouseMove . Используется при проектировании новых классов
MouseUp	Вызывает обработчик события OnMouseUp . Используется при проектировании новых классов
Refresh	Немедленно перерисовывает компонент на экране, вызывая метод Repaint
Repaint	Немедленно перерисовывает компонент на экране, вызывая при необходимости метод Invalidate
ReplaceDockedControl	Встраивает компонент на место другого уже встроенного компонента
ScreenToClient	Преобразует координаты экрана в координаты клиентской области компонента
SendCancelMode	Прерывает модальное состояние элемента
SendToBack	Переносит компонент в Z-последовательности ниже других компонентов на той же форме
SetBounds	Задаёт сразу 4 свойства: Left , Top , Width и Height
SetTextBuf	Записывает в заданный буфер значение свойства Text . Используется, если нужна обратная совместимость с 16-битными кодами
Show	Делает видимым невидимый компонент
UpdateBoundsRect	Изменяет, как и SetBounds , полное описание BoundsRect , но не перерисовывает изображение компонента на экране

Помимо перечисленных методов и многих других вспомогательных, класс **TControl** наследует методы своих классов-предков.

События

В классе **TControl**, в отличие от предшествующих ему в иерархии, описаны не только свойства и методы, но и следующие события:

Событие	Описание
OnCanResize	Событие перед началом изменения размеров компонента. Позволяет отказаться от изменения размеров или уточнить новую длину и ширину компонента
OnClick	Событие при щелчке на компоненте и некоторых других действиях пользователя
OnConstrainedResize	Происходит сразу после OnCanResize и позволяет дополнительно проконтролировать перестраивание компонента
OnDblClick	Событие при двойном щелчке на компоненте
OnDragDrop	Событие при отпуске перетаскиваемого компонента
OnDragOver	Событие при перетаскивании объекта над компонентом
OnEndDock	Событие при окончании или прерывании перетаскивания и встраивания
OnEndDrag	Событие при окончании или прерывании перетаскивания
OnMouseDown	Событие при нажатии кнопки мыши над объектом
OnMouseMove	Событие при перемещении указателя мыши над объектом
OnMouseUp	Событие при отпуске нажатой кнопки мыши над объектом
OnResize	Событие после изменения размеров компонента
OnStartDock	Событие при начале перетаскивания и встраивания объекта
OnStartDrag	Событие при начале перетаскивания объекта

TControlState — тип

См. **ControlState**

TControlStyle — тип

См. **ControlStyle**

TCursor — тип

См. **Cursor**

TCustomEdit — базовый класс окон редактирования

Абстрактный базовый класс окон редактирования

Иерархия *TObject* — *TPersistent* — *TComponent* — *TControl* — *TWinControl*

Модуль *stdctrls*

Описание

Класс **TCustomEdit** является базовым классом для окон редактирования **Edit** и **Memo**. В нем инкапсулированы основные методы и свойства, используемые при редактировании текстов. Они обеспечивают:

- Такие функции редактирования текста, как выделение фрагмента, преобразования выделенного текста, чувствительность к регистру.
- Возможность откликаться на изменения в тексте.
- Управление доступом к тексту, например, доступ «только для чтения» или символы пароля, делающие невидимыми вводимые символы.

Создавать экземпляры объектов типа **TCustomEdit** невозможно. Этот класс используется только для создания производных классов, наследующих особенности обработки текстов.

Свойства

Основные свойства класса **TCustomEdit**:

Свойство	Тип	Описание
AutoSelect	bool	Определяет, выделяется ли весь текст при получении элементом фокуса
AutoSize	bool	Определяет, будет ли высота элемента автоматически адаптироваться к размеру символов текста
BorderStyle	enum {bsNone, bsSingle};	Определяет наличие рамки вокруг окна редактирования. В комбинации с Ctl3D создает различные зрительные эффекты
CanUndo	bool	Указывает, были ли произведены в тексте операции редактирования, которые могут быть удалены командой Undo . Может использоваться для регулирования доступа к соответствующему разделу меню. Свойство только для чтения
CharCase	enum TEdit-CharCase { ecNormal, ecUpperCase, ecLowerCase };	Определяет регистр отображаемого текста: ecNormal — все символы отображаются с учетом регистра, в котором был введен каждый из них, ecUpperCase — все символы текста приводятся к верхнему регистру, ecLowerCase — все символы текста приводятся к нижнему регистру
HideSelection	bool	При значении true делает невидимым выделение части текста, когда компонент теряет фокус. В противном случае выделение остается и при потере фокуса
MaxLength	int	Устанавливает максимальную длину текста. Значение 0 указывает на неограниченную длину
Modified	bool	Указывает, был ли изменен текст
PasswordChar	char	Указывает символ, появляющийся в окне редактирования вместо вводимого пользователем символа. При значении 0 отображается вводимый символ, в противном случае — заданный символ ввода пароля

Свойство	Тип	Описание
ReadOnly	bool	Устанавливает запрет редактирования текста пользователем
SelLength	int	Указывает длину выделенного фрагмента текста
SelStart	int	Указывает первый символ выделенного фрагмента текста. Если выделения нет, то указывает символ, перед которым расположен курсор
SelText	System::Ansi-String	Содержит выделенный фрагмент текста. Можно задавать значение SelText , чтобы заменить им выделенный текст. Если выделения нет, то задание SelText приведет к вставке заданного текста в позицию курсора (SelStart)

Помимо этого имеется множество свойств, наследуемых от **TWinControl**.

Методы

Ниже приводится таблица основных методов класса **TCustomEdit**.

Метод	Описание
Clear	Удаление всего текста
ClearSelection	Удаление выделенного текста
ClearUndo	Очистка буфера, используемого для команды Undo . После этого сделанные изменения в тексте не могут быть отменены
CopyToClipboard	Копирование в буфер Clipboard выделенного текста
CutToClipboard	Вырезание в буфер Clipboard выделенного текста
PasteFromClipboard	Вставка текста из буфера Clipboard вместо выделенного текста или, если нет выделения, то вставка текста в позицию курсора
SelectAll	Выделение всего текста
Undo	Отмена результатов последней операции редактирования

События

В классе **TCustomEdit** определено только одно событие:

Событие	Описание
OnChange	Событие при попытке изменения пользователем текста. Произошло ли изменение в действительности, можно определить по значению свойства Modified

TDragMode — тип

См. **DragMode**.

TFont — тип

Определяет характеристики шрифта

Иерархия *TObject* — *TPersistent* — *TGraphicsObject*Модуль *graphics*

Описание

Объект типа **TFont** определяет множество характеристик, описывающих шрифт, используемый при отображении текстов: высоту шрифта, его имя, атрибуты (полужирный, курсив) и т.д. Используется в свойстве **Font**.

Основное свойство объекта — название шрифта **Name**. Если используется шрифт с несколькими наборами символов, то надо правильно установить свойство **CharSet** — набор символов.

Если заданная комбинация свойств **Name**, **CharSet**, **Pitch**, **Size** определяет шрифт, отсутствующий в системе, Windows подберет другой близкий шрифт.

При создании объекта **TFont** он инициализируется следующими значениями свойств: **Color** = **clWindowText**, **Name** = **MS Sans Serif**, **Size** равным 8, **Pitch** = **fpDefault**, **CharSet** = **DEFAULT_CHARSET**. Автоматически устанавливается значение **PixelsPerInch**.

Свойства

Свойство	Тип	Описание
CharSet	TFontCharset	Определяет набор символов шрифта
Color	TColor	Определяет цвет текста
FontAdapter	_di_IChangeNotifier	Интерфейс для передачи информации о шрифте в элементы ActiveX
Handle	HFONT	Дескриптор шрифта, используемый как параметр функций API Windows, требующих обработки шрифтов. Применяется только в специальных случаях
Height	int	Характеризует высоту шрифта в пикселях
Name	System::AnsiString	Вид (имя) шрифта
Pitch	enum TFontPitch { fpDefault , fpVariable , fpFixed }	Определяет способ установки ширины символов
PixelsPerInch	int	Число пикселей принтера или экрана на дюйм. Используется при копировании шрифта с канвы формы на принтер, чтобы обеспечить соответствие размеров шрифта на экране и принтере. Влияет только на печать. Изменяться пользователем не должно
Size	int	Размер шрифта в кеглях (пунктах)
Style	enum TFontStyle { fsBold , fsItalic , fsUnderline , fsStrikeOut }	Стиль шрифта — множество свойств: полужирный, курсив, подчеркнутый, перечеркнутый

Методы

TFont имеет методы, унаследованные от базовых классов, но переопределенные с учетом особенностей данного класса. Среди них можно отметить следующий:

Assign

Копирование свойств одного объекта типа **TFont** в другой объект. Свойство **PixelsPerInch** методом **Assign** не копируется. Поэтому метод можно использовать для копирования экран-ных шрифтов в шрифты принтера и наоборот

События

TFont наследует событие **OnChange** от базового класса **TGraphicsObject**.

Развернутые примеры использования и исследования типа **TFont** см. в разделе **Font**.

TGraphic — базовый класс графических объектов

Абстрактный базовый класс графических объектов типа битовых матриц, пиктограмм, метафайлов и типов, определенных пользователем

Иерархия *TObject — TPersistent*

Модуль *graphics*

Описание

Класс **TGraphic** обеспечивает производные от него классы методами хранения, манипулирования и визуализации графических объектов, методами работы с объектами типа **TPicture** и с буфером обмена Clipboard. Свойства класса **TGraphic** дают информацию о состоянии и размерах изображения.

Когда тип графики, с которой ведется работа, известен: битовая матрица, пиктограмма или метафайл, можно использовать объекты производных от **TGraphic** классов **TBitmap**, **TIcon** или **TMetafile** соответственно. Если формат графики неизвестен, то можно использовать класс **TPicture**, способный работать с графическими объектами любых типов, производных от **TGraphic**.

Свойства

Ниже приведен список свойств, определенных в **TGraphic**.

Свойство	Тип	Описание
Empty	bool	Указывает, содержит ли объект изображение. Свойство только для чтения
Height	int	Указывает высоту изображения в пикселях. Для битовых матриц может изменяться пользователем, что вызывает создание копии матрицы с указанным размером
Modified	bool	Указывает, был ли изменен графический объект. Может принимать значение true (изменен) только для битовых матриц. Для пиктограмм и метафайлов всегда равен false , даже если изменения были
Palette	HPALETTE	Управляет цветами графического изображения. Если изображение не нуждается или не имеет палитры, то Palette = 0
PaletteModified	bool	Указывает, была ли изменена палитра графического объекта. Используется в обработчиках событий OnChange
Transparent	bool	Указывает, является ли изображение прозрачным. Используется только для битовых матриц

Свойство	Тип	Описание
Width	int	Указывает ширину изображения в пикселях. Для битовых матриц может изменяться пользователем, что вызывает создание копии матрицы с указанным размером

Методы

Ниже приведены основные методы, объявленные или переопределенные в классе **TGraphic**.

Метод	Описание
<u>Assign</u>	Копирует изображение из другого графического объекта, в частности, из буфера обмена Clipboard
<u>LoadFromClipboardFormat</u>	Читает изображение из буфера обмена Clipboard в заданном формате
<u>LoadFromFile</u>	Читает изображение из файла
<u>LoadFromStream</u>	Читает графическое изображение из указанного потока
<u>SaveToClipboardFormat</u>	Сохраняет изображение в буфере обмена Clipboard в заданном формате
<u>SaveToFile</u>	Сохраняет изображение в файле
<u>SaveToStream</u>	Записывает изображение в поток

События

Событие	Описание
<u>OnChange</u>	Событие при изменении графического объекта
<u>OnProgress</u>	События происходят при медленных процессах изменения графического изображения и позволяют построить индикатор хода процесса

TIcon — класс

Инкапсулирует пиктограмму Windows

Иерархия *TObject* — *TPersistent* — *TGraphics*

Модуль *graphics*

Описание

Класс **TIcon** инкапсулирует пиктограмму Windows. Свойства класса **TIcon** имеют такие объекты, как **TForm** и **TPicture**.

Пиктограмма соответствует формату файлов **.ico** Windows. Рисовать пиктограммы на канве можно методом канвы **Draw**, но метод **StretchDraw** к ним не применим, поскольку пиктограммы не могут изменять своих размеров.

Свойства

Ниже приведен список основных свойств, определенных или переопределенных в **TIcon**.

Свойство	Тип	Описание
Empty	bool	Указывает, содержит ли объект пиктограмму. Свойство только для чтения
Handle	HICON	Обеспечивает доступ к обработке пиктограмм в GDI Windows. Используется при вызовах функций API Windows
Height	int	Указывает высоту изображения в пикселях. Размеры всех пиктограмм в приложении одинаковы и задаются установками Windows. Попытка изменить Height ведет к генерации исключения
Transparent	bool	Указывает, является ли изображение прозрачным. Может использоваться только для чтения. Попытка изменить Transparent ведет к генерации исключения
Width	int	Указывает ширину изображения в пикселях. Размеры всех пиктограмм в приложении одинаковы и задаются установками Windows. Попытка изменить Width ведет к генерации исключения

Методы

Ниже приведены основные методы, объявленные или переопределенные в классе **TIcon**.

Метод	Описание
<u>Assign</u>	Копирует изображение из другого графического объекта, в частности, из буфера обмена Clipboard
<u>LoadFromClipboardFormat</u>	Читает изображение из буфера обмена Clipboard в заданном формате
<u>LoadFromFile</u>	Читает изображение из файла
<u>LoadFromStream</u>	Читает графическое изображение из указанного потока
<u>ReleaseHandle</u>	Возвращает дескриптор типа HIcon и устанавливает дескриптор объекта TIcon в NULL
<u>SaveToClipboardFormat</u>	Сохраняет изображение в буфере обмена Clipboard в заданном формате
<u>SaveToFile</u>	Сохраняет изображение в файле
<u>SaveToStream</u>	Записывает изображение в поток

События

Событие	Описание
<u>OnChange</u>	Событие при изменении графического объекта
<u>OnProgress</u>	События происходят при медленных процессах изменения графического изображения и позволяют построить индикатор хода процесса

TImageList — компонент и класс

Список, использующийся для хранения набора изображений одинаковых размеров, на которые можно ссылаться по индексам

Иерархия *TObject — TPersistent — TComponent — TCustomImageList — TDragImageList*

Модуль *controls*

Описание

TImageList используется и как компонент, и как свойство других компонентов, например, свойство **Images** компонента типа **TMainMenu**. Представляет собой набор изображений одинаковых размеров, на которые можно ссылаться по индексам, начинающимся с 0. Используется для эффективного управления множеством пиктограмм и битовых матриц. Может включать в себя монохромные битовые матрицы, содержащие маски для прозрачности рисуемых изображений.

Изображения в компонент **TImageList** могут быть загружены в процессе проектирования двойным щелчком на компоненте и последующим выбором из файлов.

Свойства

Основные свойства, наследуемые **TImageList** (новых свойств в классе не объявлено):

Свойство	Тип	Описание
AllocBy	int	Определяет количество изображений, на которое увеличивается список для добавления новых изображений
BkColor	Graphics::TColor	Определяет цвет фона, используемый при маскировании области изображений. Если BkColor равен clNone , изображение рисуется прозрачным с использованием маски. В противном случае BkColor определяет цвет вне маски. BkColor не влияет на изображение, если свойство Masked установлено в false
BlendColor	Graphics::TColor	Определяет цвет фона, используемый при рисовании изображения. Это цвет, который комбинируется с указанным цветом при значениях DrawingStyle , равных dsFocus и dsSelected . Значение clNone соответствует отсутствию комбинируемого цвета, а значение clDefault означает системный цвет выделения
Count	int	Определяет число изображений в списке. Свойство только для чтения
DrawingStyle	enum TDrawingStyle {dsFocus, dsSelected, dsNormal, dsTransparent}	Указывает стиль, используемый при рисовании
Handle	int	Дескриптор, используемый при вызовах функций API Windows

Свойство	Тип	Описание
Height	int	Высота изображений в списке
ImageType	enum TImageType {itImage, itMask}	Определяет, использует ли список изображение или маску
Masked	bool	Определяет, содержит ли список маски, комбинируемые с изображениями
Width	int	Ширина изображений в списке

События

В **TImageList** определено только одно событие:

Событие	Описание
OnChange	Происходит при изменении списка

TList — класс

Содержит список указателей на любые объекты

Иерархия *TObject*

Модуль *classes*

Описание

Объект типа **TList** предназначен для хранения и управления списком указателей на объекты. Свойства и методы **TList** позволяют добавлять и удалять элементы списка, изменять их расположение в списке, сортировать элементы и проводить другие манипуляции с данными.

Свойства

Свойство	Тип	Описание
Capacity	int	Число указателей, которые могут храниться в объекте
Count	int	Число указателей, хранящихся в объекте
Items	void *	Элементы массива указателей в объекте
List	void *	Указатель на массив указателей в объекте

Остальные свойства наследуются от базового класса **TObject**.

Методы

Метод	Описание
Add	Добавление нового указателя в список
Clear	Очистка списка
Delete	Удаление из списка элемента по его индексу
Error	Генерация исключения
Exchange	Взаимная перестановка двух элементов списка
Expand	Расширение емкости списка и копирование списка

Метод	Описание
<u>Add</u>	Добавление нового указателя в список
<u>First</u>	Возвращение первого указателя списка
<u>IndexOf</u>	Определение первое вхождение в список заданного указателя
<u>Insert</u>	Вставка элемента в список в заданную позицию
<u>Last</u>	Возвращение последнего указателя списка
<u>Move</u>	Изменение текущей позиции элемента в списке на заданную
<u>Pack</u>	Удаление из списка указателей NULL
<u>Remove</u>	Удаление из списка элемента по его значению
<u>Sort</u>	Сортировка списка

Остальные методы наследуются от базового класса **TObject**.

Пример

Ниже приведен пример использования объекта типа **TList** для создания в памяти временной базы данных из последовательности записей, содержащих фамилию и год рождения сотрудника. Особенности использования отдельных свойств и методов смотрите в соответствующих справочных разделах.

```
// Определение типа структуры
typedef struct
{
    AnsiString Name;
    Word Year;
} Rec;
// Объявление указателя на структуру
Rec * PRec;

TList *MyList = new TList; // Создается список
PRec = new Rec;           // Выделяется место под запись
PRec->Name = "Петров";     // Заполняются поля записи
PRec->Year = 1960;
MyList->Add(PRec);         // Указатель на запись заносится в список
// Те же операции с новой записью
PRec = new Rec;
PRec->Name = "Иванов";
PRec->Year = 1950;
MyList->Add(PRec);
....
// Изменение года рождения - доступ к структуре через список
((Rec *)MyList->Items[1])->Year = 1970;
....
// Поиск в записях самого молодого сотрудника
PRec = (Rec *)MyList->Items[0];
for(int i = 1; i < MyList->Count; i++)
    if(((Rec *)MyList->Items[i])->Year > PRec->Year)
        PRec = (Rec *)MyList->Items[i];
ShowMessage("Самый молодой - " + PRec->Name);
....
// Освобождение памяти
for(int i = 0; i < MyList->Count; i++)
    delete (Rec *)MyList->Items[i];
delete MyList;
```

TMetafile — класс

Инкапсулирует метафайл Win32 Enhanced

Иерархия *TObject* — *TPersistent* — *TGraphic*

Модуль *graphics*

Описание

Класс **TMetafile** позволяет хранить изображения, соответствующие графическим метафайлам **.emf** Windows. Свойства **TMetafile** описывают размер и характеристики метафайла. Рисовать метафайлы на канве можно методами канвы **Draw** и **StretchDraw**. Свойство **Enhanced** определяет, как метафайл хранится на диске: **true** соответствует формату **.emf** (Win32 Enhanced Metafile), а **false** — **.wmf** (Windows 3.1 Metafile).

Свойства

Ниже приведен список основных свойств, определенных или переопределенных в **TMetafile**.

Свойство	Тип	Описание
CreatedBy	AnsiString	Указывает имя автора или приложения, создавшего метафайл. Свойство только для чтения. Чтобы задать CreatedBy нового метафайла, надо вызвать конструктор объекта TMetafileCanvas , предусматривающий задание CreatedBy
Description	AnsiString	Определяет не обязательный текст описания метафайла. Свойство только для чтения. Чтобы задать Description нового метафайла, надо вызвать конструктор объекта TMetafileCanvas , предусматривающий задание Description
Empty	bool	Указывает, содержит ли объект метафайл. Свойство только для чтения
Enhanced	bool	Значение true соответствует формату хранения на диске .emf , а false — .wmf . В памяти метафайл всегда хранится в формате .emf . Формат .wmf ведет к частичной потере информации и оставлен только для обратной совместимости с Windows 3.x
Handle	int	Дескриптор, используемый для доступа к GDI Windows и вызова функций API Windows
Height	int	Указывает высоту изображения в пикселях. Может изменяться пользователем
Inch	Word	Используется для метафайлов WMF и указывает число единиц на дюйм, необходимое для масштабирования. Метафайлы EMF хранят эту информацию внутри себя
MMHeight	int	Содержит высоту изображения в единицах 0.01 мм. MMHeight дает более точное значение высоты, чем свойство Height , значение которого измеряется в пикселях

Свойство	Тип	Описание
MMWidth	int	Содержит ширину изображения в единицах 0.01 мм. MMWidth дает более точное значение ширины, чем свойство Width , значение которого измеряется в пикселях
Modified	bool	Указывает, был ли изменен графический объект
Palette	HPALETTE	Управляет цветами графического изображения. Если изображение не нуждается или не имеет палитры, то Palette = 0
PaletteModified	bool	Указывает, была ли изменена палитра графического объекта. Используется в обработчиках событий OnChange
<u>Transparent</u>	bool	Указывает, является ли изображение прозрачным
Width	int	Указывает ширину изображения в пикселях. Может изменяться пользователем

Методы

Ниже приведены основные методы, объявленные или переопределенные в классе **TMetafile**.

Метод	Описание
<u>Assign</u>	Копирует изображение из другого графического объекта, в частности, из буфера обмена Clipboard
Clear	Удаление изображения
<u>LoadFromClipboardFormat</u>	Читает изображение из буфера обмена Clipboard в заданном формате
<u>LoadFromFile</u>	Читает изображение из файла
<u>LoadFromStream</u>	Читает графическое изображение из указанного потока
ReleaseHandle	Возвращает дескриптор объекта и устанавливает его дескриптор в NULL
<u>SaveToClipboardFormat</u>	Сохраняет изображение в буфере обмена Clipboard в заданном формате
<u>SaveToFile</u>	Сохраняет изображение в файле
<u>SaveToStream</u>	Записывает изображение в поток

События

Событие	Описание
<u>OnChange</u>	Событие при изменении графического объекта
<u>OnProgress</u>	События происходят при медленных процессах изменения графического изображения и позволяют построить индикатор хода процесса

TObject — базовый класс всех объектов

Базовый класс всех объектов в C++Builder

Модуль *systobj*

Описание

Класс **TObject** инкапсулирует основные функции, свойственные всем объектам C++Builder. Интерфейс **TObject** обеспечивает:

- Возможность создания, управления и разрушения экземпляров объектов, включая выделение под них памяти, инициализацию и освобождение памяти после их уничтожения.
- Поддержка информации об объектах и типах (run-time type information — RTTI).
- Поддержка обработки сообщений.

Все классы в C++Builder являются прямыми или косвенными наследниками **TObject**. Прямое наследование используется только при объявлении простых классов, объекты которых не являются компонентами, не могут присваиваться друг другу и не участвуют в операциях обмена с потоками. Подавляющее большинство классов являются косвенными наследниками **TObject** и производятся от промежуточных классов. Если при объявлении нового типа объектов не указывается класс-предок, то C++Builder считает **TObject** предком нового класса.

Большинство методов **TObject** не используются непосредственно в компонентах, с которыми имеет дело пользователь. Исходные методы **TObject** обычно перегружены в классах-наследниках или заменены другими, построенными на их основе.

Хотя формально **TObject** не является абстрактным классом, но объекты этого класса создавать нельзя.

Методы

Большинство методов класса **TObject** пользователями непосредственно не используется. Ниже приводятся только те методы, прямое обращение к которым может оказаться полезным или которые могут использоваться при создании новых классов.

Метод	Описание
ClassName	Возвращает имя типа объекта
ClassNameIs	Возвращает true , если передаваемое в функцию имя совпадает с именем данного класса
ClassParent	Возвращает тип непосредственного предка данного класса.
Create	Конструктор. Не осуществляет инициализацию каких-то данных, так что перегружается в классах-наследниках.
Free	Уничтожает объект и освобождает выделенную под него память. Вызывать этот метод не нужно — он вызывается автоматически при использовании ключевого слова delete
FreeInstance	Освобождает память, выделенную ранее вызванным методом NewInstance . Автоматически вызывается деструктором объекта. Непосредственный вызов пользователем не требуется. Должен быть перегружен, если перегружен метод NewInstance . Использует InstanceSize для определения размера выделенной области памяти

Метод	Описание
InheritsFrom	Определяет, является ли указанный класс предком данного объекта
InitInstance	Инициализирует новый объект и указатель на его таблицу виртуальных методов. Вызывается автоматически методом NewInstance . Не может быть перегружен
InstanceSize	Возвращает число байтов, занимаемых объектом. Может использоваться для анализа затрат памяти различными типами объектов
NewInstance	Выделяет область памяти под объект и возвращает указатель на нее. Автоматически вызывается всеми конструкторами. Использует InstanceSize . Может перегружаться в классах-наследниках

TPen — тип

Определяет свойства пера, используемые при рисовании линий и фигур на канве

Иерархия *TObject* — *TPersistent* — *TGraphicsObject*

Модуль *graphics*

Описание

Тип **TPen** инкапсулирует атрибуты пера Windows при рисовании на канве — объекте типа **TCanvas**.

Свойства

Свойство	Тип	Описание
Color	TColor	Цвет пера. По умолчанию — clBlack
Handle	HPEN	Дескриптор пера окна. Используется для доступа к функциям API Windows
Mode	TPenMode	Определяет режим рисования линий
Style	TPenStyle	Определяет стиль рисования линий
Width	int	Определяет толщину линии в пикселях. Влияет на Style

Методы

В классе **TPen** не введено каких-то принципиально новых методов. Переопределены такие общие методы, как **Assign**, конструктор и деструктор. Остальные методы наследуются от классов-предков.

События

Класс **TPen** наследует от класса **TGraphicsObject** событие **OnChange**, наступающее после изменения графического объекта.

TPersistent — базовый класс объектов, участвующих в операциях с потоками

Базовый класс всех объектов C++Builder, допускающих операцию присваивания или участвующих своими свойствами в операциях с потоками

Иерархия *TObject*

Модуль *classes*

Описание

Класс **TPersistent** инкапсулирует фундаментальные свойства объектов, которые должны иметь возможность присваиваться друг другу или читать и записывать свои свойства в поток. Методы класса **TPersistent** обеспечивают:

- Определение процедур загрузки и сохранения данных в потоке.
- Присваивание значений свойствам.
- Присваивание содержимого одного объекта другому.

Объекты **TPersistent** создаваться не могут. Класс используется только для создания производных классов.

Методы

Класс имеет следующие методы:

Метод	Описание
Assign	Копирует данные одного объекта в другой
AssignTo	Защищенная реализация метода, аналогичного Assign
DefineProperties	Виртуальный защищенный метод обмена данными с потоком
GetNamePath	Возвращает строку, используемую в Инспекторе Объектов
GetOwner	Защищенный метод, возвращающий владельца объекта

Из всех этих методов только **Assign** может непосредственно использоваться пользователем при работе с объектами. Остальные могут потребоваться только при создании новых классов.

TPicture — класс

Описывает объект, являющийся контейнером графических объектов типа битовых матриц, пиктограмм, метафайлов и определенных пользователем типов графических объектов

Иерархия *TObject* — *TPersistent*

Модуль *graphics*

Описание

Объект типа **TPicture** является контейнером любого графического объекта **TGraphic**, тип которого указывается свойством **Graphic**. Объект **TPicture** имеет полиморфные методы файлового чтения и записи **LoadFromFile** и **SaveToFile**, автоматически подстраивающиеся под тип объекта.

В зависимости от типа хранимого объекта — битовой матрицы, пиктограммы, метафайла, определены соответствующие свойства **Bitmap**, **Icon** или **Metafile**, указывающие на графический объект. При ошибочном обращении к этим свойствам (если объект в действительности имеет другой тип) прежнее содержимое объекта стирается и открывается новый пустой объект указанного типа. Вместо этих свойств можно получать доступ к графическому объекту непосредственно через свойство **Graphic**. Таким образом, например, обращения **Image1->Picture->Graphic** и **Image1->Picture->Bitmap** эквивалентны, если графический объект — битовая матрица.

Обмен объекта **TPicture** с буфером обмена Clipboard может осуществляться методом **Assign** объекта **TClipboard**.

Свойства

В классе **TPicture** объявлены следующие основные свойства:

Свойство	Тип	Описание
Bitmap	<u>TBitmap</u>	Указывает на хранящийся объект как на битовую матрицу (формат файла .bmp)
Graphic	<u>TGraphic</u>	Указывает на хранящийся объект как на битовую матрицу, пиктограмму, метафайл или определенный пользователем тип
Height	int	Указывает собственную, не измененную высоту изображения в пикселях. Свойство только для чтения
Icon	<u>TIcon</u>	Указывает на хранящийся объект как на пиктограмму (формат файла .ico)
Metafile	<u>TMetafile</u>	Указывает на хранящийся объект как на метафайл (формат файла .emf)
Width	int	Указывает собственную, не измененную ширину изображения в пикселях. Свойство только для чтения

Помимо описанных класс наследует множество свойств родительских классов.

Методы

Ниже приведены основные методы, объявленные в классе **TPicture**.

Метод	Описание
<u>LoadFromClipboardFormat</u>	Читает изображение из буфера обмена Clipboard в заданном формате
<u>LoadFromFile</u>	Читает изображение из файла
RegisterClipboardFormat	Регистрирует новый формат Clipboard графического объекта для использования в методе <u>LoadFromClipboardFormat</u>
RegisterFileFormat	Регистрирует новый файловый формат графического объекта для использования в методе <u>LoadFromFile</u>
RegisterFileFormatRes	Регистрирует новый файловый формат графического объекта из ресурсов для использования в методе <u>LoadFromFile</u>
<u>SaveToClipboardFormat</u>	Сохраняет изображение в буфере обмена Clipboard в заданном формате
<u>SaveToFile</u>	Сохраняет изображение в файле
SupportsClipboardFormat	Определяет, поддерживается ли указанный формат Clipboard графического объекта
UnregisterGraphicClass	Удаляет все ссылки на ранее зарегистрированный формат Clipboard или файла

События

Событие	Описание
OnChange	Событие при изменении графического объекта
OnProgress	События происходят при медленных процессах изменения графического изображения и позволяют построить индикатор хода процесса

TPoint — тип

Определяет точку координат в пикселях

Модуль *Windows*

Определение (упрощенное)

```
struct TPoint
{
    int  x;
    int  y;
};
```

Описание

Тип **TPoint** определяет точку координат в пикселях. В частности, во многих свойствах этот тип используется для задания координат углов прямоугольников и других фигур. В этих случаях началом координат, в зависимости от применения, считается левый верхний угол экрана или окна.

Для задания параметров типа **TPoint** во многих функциях удобно использовать функцию **Point** (см. раздел 15.3.3 главы 15), возвращающую структуру **TPoint**.

Выше приведено упрощенное определение, содержащее только имена полей. Полной определение вы можете посмотреть во встроенной справке C++Builder или в файле *Windows.cpp*.

TRect — тип

Определяет координаты прямоугольной области

Модуль *Windows*

Определение (упрощенное)

```
struct TRect
{
    int Left;
    int Top;
    int Right;
    int Bottom;
};
```

Описание

Координаты задаются и как четыре целых числа, определяющих координаты в пикселях левой, верхней, правой и нижней сторон прямоугольника, и как две точки типа **TPoint**, представляющие собой координаты левого верхнего и правого нижнего углов. Началом координат обычно считается левый верхний угол экрана или окна.

Для типа определены функции-элементы **Width()** и **Height()**, возвращающие соответственно ширину и высоту. Определены также операции «==» и «!=».

Задавать значения переменной типа **TRect** удобно функцией **Rect**, задающей координаты левой, верхней, правой и нижней границ — **Left**, **Top**, **Right**, **Bottom**,

или функцией **Bounds**, задающей координаты левого верхнего угла **Left** и **Top**, ширину прямоугольника **AWidth** и его высоту **AHeight**. См. описания этих функций и примеры в главе 15 в разделе 15.3.3.

TStringFloatFormat — тип

В ряде методов тип **TStringFloatFormat** определяет формат представления чисел строкой

Определение

```
enum TStringFloatFormat {sffGeneral, sffExponent, sffFixed,
                        sffNumber, sffCurrency };
```

Различные значения формата означают следующее:

Значение	Описание
sffGeneral	Значение преобразуется в наиболее компактное из двух форматов: с фиксированной точкой или научного формата. Младшие нулевые разряды усекаются. Десятичная точка появляется только при необходимости. Формат с фиксированной точкой используется только при числе цифр целой части больше не больше указанной точности и при значениях не меньше 0.00001. В остальных случаях используется научный формат с минимальным числом цифр в степени порядка (от 0 до 4).
sffExponent	Научный формат. Значение преобразуется в строку вида «-d.ddd...E+dddd». Символ '-' записывается только для отрицательных чисел. Перед десятичной точкой записывается всегда одна цифра. Общее число цифр (включая цифру перед точкой) определяется заданной точностью. После символа 'E' всегда ставится знак + или -. Число цифр в степени (порядок числа) лежит в пределах от 0 до 4.
sffFixed	Формат с фиксированной точкой. Значение преобразуется в строку вида «-ddd.ddd...». Символ '-' записывается только для отрицательных чисел. Перед десятичной точкой записывается по крайней мере одна цифра. Число цифр после точки определяется заданным числом разрядов (от 0 до 18). Если число цифр слева от точки должно быть больше заданной точности, используется научный формат.
sffNumber	Числовой формат. Значение преобразуется в строку вида «-d,ddd,ddd.ddd...». Совпадает с форматом sffFixed за исключением наличия разделителей после каждых трех разрядов в целой части.
sffCurrency	Монетарный формат для представления чисел, отображающих денежные суммы. Определяется установками Windows (глобальными переменными CurrencyString , CurrencyFormat , NegCurrFormat , ThousandSeparator , DecimalSeparator). Число цифр после десятичной точки определяется заданным числом разрядов (от 0 до 18).

Для всех форматов действительные символы, используемые в качестве десятичной точки и разделителя тысяч определяются глобальными переменными **DecimalSeparator** и **ThousandSeparator**.

TStringList — класс

Список строк с расширенными возможностями манипулирования ими

Иерархия *TObject* — *TPersistent* — *TStrings*

Модуль *classes*

Описание

Класс **TStringList** наследует классу **TStrings**, реализуя многие его абстрактные свойства и методы и вводя некоторые новые возможности:

- сортировку строк в списке
- запрещение хранения дубликатов строк
- реакцию на изменения содержания списка

Свойства

Ниже приведен список основных свойств, определенных в **TStringList**.

Свойство	Тип	Описание
<u>Capacity</u>	int	Указывает число строк, которые может содержать список, позволяет заранее выделить память для добавления нескольких строк
<u>Count</u>	int	Число строк в списке. Свойство только для чтения
<u>Objects</u> [int Index]	System::TObject *	Возвращает объект, связанный с указанной строкой свойства Strings
<u>Duplicates</u>	enum Tduplicates { dupIgnore, dupAccept, dupError }	Указывает, могут ли добавляться в сортированный список дубликаты строк. Значение dupIgnore — игнорирование добавления дубликата, dupAccept — разрешение добавления дубликата, dupError — генерация исключения EListError при попытке добавления дубликата строки. Значения dupAccept и dupError никак не реагируют на уже имеющиеся в списке дубликаты. На несортированный список свойство Duplicates не оказывает никакого влияния
<u>Sorted</u>	bool	Указывает, должны ли строки в списке автоматически сортироваться по алфавиту
<u>Strings</u> [int Index]	System::AnsiString	Текст строки с указанным индексом. Индекс первой строки — 0

Кроме того **TStringList** наследует от **TStrings** такие свойства, как **CommaText**, **Names**, **StringsAdapter**, **Text**, **Values**.

Методы

Класс **TStringList** наследует от **TStrings** такие свойства, **Add**, **Clear**, **Delete**, **Exchange**, **IndexOf**, **Insert** и много других. Кроме того в классе **TStringList** объявлены методы:

Метод	Описание
bool Find(const System::AnsiString S, int &Index)	Определяет, имеется ли заданная строка S в сортированном списке, и, если имеется, то возвращает в параметр Index индекс этой строки. Для не сортированных списков следует использовать метод IndexOf
Sort	Сортирует строки списка, свойство Sorted которого установлено в false , в возрастающей алфавитной последовательности. Если Sorted = true , то список сортируется автоматически

TStrings — класс

Абстрактный класс объектов, представляющих собой списки строк и используемых во многих компонентах C++Builder в качестве различных свойств

Иерархия *TObject* — *TPersistent*

Модуль *classes*

Описание

Класс **TStrings** содержит методы и свойства, позволяющие манипулировать со списками строк:

- Добавлять и удалять строки в указанных позициях
- Перестраивать и упорядочивать последовательность строк
- Получать доступ к конкретным строкам
- Читать и записывать списки строк в файлы и потоки
- Связывать с каждой строкой некоторый объект

Свойства

Ниже приведен список основных свойств, определенных в **TStrings**.

Свойство	Тип	Описание
Capacity	int	Указывает число строк, которые может содержать список. В классе TStrings чтение Capacity возвращает значение Count , а запись значения Capacity ничего не изменяет в списке. Но в некоторых классах, производных от TStrings , свойство Capacity позволяет заранее выделить память для добавления нескольких строк
CommaText	System::AnsiString	Возвращает текст, в котором отдельные строки объединены в одну строку формата SDF (system data format). Отдельные исходные строки разделяются в итоговой строке запятыми и каждая строка, если в ней имеются символы пробелов, заключается в двойные кавычки. Если в исходных строках использовались символы двойных кавычек, они дублируются (получается два следующих друг за другом символа)
Count	int	Число строк в списке. Свойство только для чтения

Свойство	Тип	Описание
Names [int Index]	System::AnsiString	Применяется для списков, имеющих структуру «Имя = Значение». Такую структуру имеют, например, файлы .ini . Свойство Names возвращает Имя, использованное в строке с указанным индексом. Если строка не имеет форму «Имя = Значение», возвращается пустая строка. Свойство только для чтения
Objects [int Index]	TObject *	Возвращает объект, связанный с указанной строкой свойства Strings . В классе TStrings свойство Objects не используется, но может использоваться в некоторых классах, производных от TStrings
Strings [int Index]	System::AnsiString	Текст строки с указанным индексом. Индекс первой строки — 0
Text	System::AnsiString	Представляет весь список как одну строку, внутри которой используются разделители типа символов возврата каретки и перевода строки
Values [System::AnsiString Name]	System::AnsiString	Применяется для списков, имеющих структуру «Имя = Значение». Такую структуру имеют, например, файлы .ini . Свойство Values возвращает Значение, в строке с указанным именем Name . Если заданное имя Name не найдено, возвращает пустая строка

Методы

Ниже приведены основные методы, объявленные в классе **TStrings**.

Метод	Описание
int Add (const System::AnsiString S)	Добавляет строку S в конец списка. Возвращает индекс добавленной строки. В окнах редактирования добавляемая строка может оказаться разбитой на несколько, так что в этих случаях возвращаемый индекс ни о чем не говорит
int AddObject (const System::AnsiString S, System::TObject* AObject)	Добавляет в список строку S и связанный с ней объект AObject . Возвращает индекс добавленной строки и объекта
void AddStrings (TStrings* Strings)	Добавляет в список группу строк Strings типа TStrings
void Append (const System::AnsiString S)	Добавляет строку в конец списка. Метод аналогичен Add , но не возвращает индекс строки
void Clear (void)	Очищает список
void Delete (int Index)	Удаляет из списка указанную строку с индексом Index
bool Equals (TStrings* Strings)	Сравнивает данный список с заданным списком Strings . Возвращает true при идентичности списков

Метод	Описание
void Exchange(int Index1, int Index2)	Переставляет местами строки списка с индексами Index1 и Index2
char * GetText (void)	Возвращает буфер, под который выделяет память, и заполняет его значением свойства Text
int IndexOf(const System::AnsiString S)	Возвращает индекс указанной строки S . Если такой строки нет в списке, возвращается -1
int IndexOfName(const System::AnsiString Name)	Применяется для списков, имеющих структуру «Имя = Значение». Такую структуру имеют, например, файлы .ini . Возвращается индекс строки, в которой имя равно заданному значению Name . Если такой строки нет в списке, возвращается -1
int IndexOfObject(System::TObject* AObject)	Возвращает индекс первой строки, связанной с заданным объектом AObject . Если такой строки нет в списке, возвращается -1
void Insert(int Index, const System::AnsiString S)	Вставляет указанную строку S в заданную позицию Index . Если Index = 0, строка вставляется в первую позицию
void InsertObject(int Index, const System::AnsiString S, System::TObject* AObject)	Вставляет указанную строку S в заданную позицию Index и связывает с ней объект AObject . Если Index = 0, строка вставляется на первую позицию
void LoadFromFile(const System::AnsiString FileName)	Заполняет список строками текста из указанного файла FileName
void LoadFromStream(TStream* Stream)	Заполняет список строками текста из указанного потока Stream
void Move(int CurIndex, int NewIndex)	Изменяет позицию строки с индексом CurIndex , давая ей индекс NewIndex
void SaveToFile(const System::AnsiString FileName)	Сохраняет строки списка в файле с указанным именем FileName
void SaveToStream(TStream* Stream)	Сохраняет значение свойства Text в указанном потоке Stream
void SetText(char * Text)	Задаёт значение свойства Text

TWinControl — базовый класс оконных компонентов

Абстрактный базовый класс всех оконных компонентов

Иерархия *TObject* — *TPersistent* — *TComponent* — *TControl*

Модуль *controls*

Описание

Класс **TWinControl** является базовым классом для всех оконных компонентов C++Builder, т.е. для компонентов, которые:

- Могут получать фокус во время выполнения приложения. Другие компоненты могут отображать данные, но пользователь не может общаться с компонентом с помощью клавиатуры, если это не оконный компонент.

- Могут содержать другие компоненты, т.е. быть компонентами-контейнерами, компонентами-родителями других, дочерних компонентов.
- Имеют дескрипторы окна.

Новые компоненты редко создаются непосредственно на основе **TWInControl**. Обычно они основываются на производных классах, таких, как **TCustomControl**, имеющий канву и обработку сообщений прорисовки, или на более специализированных классах типа **TButtonControl**, **TCustomComboBox**, **TCustomEdit** или **TCustomListBox**.

Свойства

Ниже приведен список основных свойств, определенных или переопределенных в **TWInControl**. Некоторые методы, используемые в основном при разработке сложных новых классов, в него не включены.

Свойство	Тип	Описание
Brush	TBrush *	Определяет цвет и стиль заполнения фона окна. Свойство только для чтения
ClientOrigin	TPoint	Экранные координаты левого верхнего угла клиентской области компонента. Свойство только для чтения
ClientRect	TRect	Содержит размер клиентской области компонента. Свойство только для чтения
ControlCount	int	Число дочерних компонентов данного оконного элемента. Свойство только для чтения
Controls [int Index]	TControl *	Массив дочерних компонентов оконного элемента. Свойство только для чтения
Ctl3D	bool	Определяет, будет ли компонент выглядеть объемным или плоским
DockClientCount	int	Число компонентов, встроенных в данный оконный элемент. Свойство только для чтения
DockClients [int Index]	TControl *	Массив компонентов, встроенных в данный оконный элемент. Свойство только для чтения
Handle	HWND	Дескриптор оконного элемента, используемый при вызове функций API Windows. Свойство только для чтения
HelpContext	THelpContext	Номер контекстно-зависимой встроенной справки
ImeMode и Ime-Name	TImeMode и System::AnsiString	Определяют обработку символов в редакторах. Обычно значения по умолчанию не изменяют
ParentCtl3D	bool	Управляет наследованием родительского свойства Ctl3D
ParentWindow	HWND	Дескриптор родительского окна, не являющегося визуальным компонентом, например TActiveXControl . Если свойство Parent не NULL, то значение ParentWindow игнорируется

Свойство	Тип	Описание
<u>Showing</u>	bool	Определяет, виден ли компонент в данный момент. Свойство только для чтения
<u>TabOrder</u>	TTabOrder	Указывает позицию компонента в последовательности табуляции
<u>TabStop</u>	bool	Определяет, может ли пользователь перевести фокус на компонент клавишей табуляции
<u>WindowHandle</u>	HWND	То же, что Handle , но это свойство можно читать и изменять

Класс **TWInControl** наследует также много свойств своих предшественников.

Методы

Ниже приведены основные методы, наследуемые от **TWInControl** и используемые в компонентах — потомках этого класса.

Метод	Описание
<u>CanFocus</u>	Определяет, может ли компонент получать фокус, т.е. получать сообщения пользователя
<u>ChangeScale</u>	Изменяет масштаб компонента и его дочерних компонентов
<u>ContainsControl</u>	Определяет, является ли указанный компонент прямым или косвенным наследником данного оконного элемента
<u>ControlAtPos</u>	Возвращает дочерний компонент, находящийся в указанной позиции
<u>DisableAlign</u>	Временно запрещает выравнивание компонентов в оконном элементе
<u>EnableAlign</u>	Отменяет действие предварительно вызванного метода <u>DisableAlign</u> и вызывает <u>Realign</u> для выравнивания компонентов
<u>FindChildControl</u>	Возвращает указатель на дочерний компонент с заданным именем или NULL
<u>FindNextControl</u>	Возвращает очередной оконный компонент в последовательности табуляции
<u>Focused</u>	Определяет, находится ли оконный элемент в фокусе
<u>GetTabOrderList</u>	Строит список дочерних компонентов в последовательности табуляции
<u>HandleAllocated</u>	Проверяет наличие дескриптора окна компонента
<u>HandleNeeded</u>	Создает дескриптор окна, если он до этого не существовал
<u>Invalidate</u>	Сообщает о необходимости перерисовки компонентов
<u>Realign</u>	Выравнивает компоненты в оконном элементе
<u>Repaint</u>	Перерисовывает изображение компонента на экране с помощью <u>Invalidate</u>

Метод	Описание
<u>ScaleBy</u>	Масштабирует оконный элемент и все содержащиеся в нем компоненты
<u>ScaleControls</u>	Изменяет масштаб компонентов в оконном элементе, не изменяя масштаба самого оконного элемента
<u>ScrollBy</u>	Сдвигает содержимое оконного элемента
<u>SelectFirst</u>	Передает фокус дочернему компоненту, первому в последовательности табуляции
<u>SelectNext</u>	Передает фокус компоненту, следующему в последовательности табуляции
<u>SetBounds</u>	Задаёт координаты и размеры элемента
<u>SetChildOrder</u>	Изменяет позицию компонента в списке дочерних компонентов
<u>SetFocus</u>	Передает фокус элементу (активизирует его)
<u>SetZOrder</u>	Перемещает компонент на верх или в них Z-последовательности
<u>Update</u>	Немедленная перерисовка компонента

Помимо перечисленных методов имеется еще немало методов, наследуемых от предков.

События

Событие	Описание
OnDockDrop	Событие при завершении перетаскивания и встраивания
OnDockOver	Событие при перемещении перетаскиваемого и встраиваемого компонента над контейнером
OnEnter	Событие при получении элементом фокуса
OnExit	Событие при потере элементом фокуса
OnGetSiteInfo	Событие происходит перед OnDockDrop и позволяет получить информацию о перетаскиваемом объекте
OnKeyDown	Событие при нажатии любой клавиши или кнопки мыши
OnKeyPress	Событие при нажатии клавиши символа
OnKeyUp	Событие при отпускании клавиши
OnMouseWheel	Событие при вращении колесика мыши
OnMouseWheelDown	Событие при вращении колесика мыши вниз. Происходит, если отсутствует обработчик OnMouseWheel
OnMouseWheelUp	Событие при вращении колесика мыши вверх. Происходит, если отсутствует обработчик OnMouseWheel
OnUnDock	Событие при попытке вывести встроенный компонент из его контейнера

16.5 Предметный указатель разделов книги, содержащих описания компонентов библиотеки VCL

Ниже приведены ссылки на те разделы книги, в которых не просто содержится упоминание того или иного компонента, но дается какая-то новая информация о компоненте или о способах его использования в приложениях. Например, метки и кнопки применяются в любом примере данной книги, но в таблице указаны только те разделы, в которых об этих компонентах сообщается что-то новое.

ActionList	3.9.1, 3.9.3, 4.1.6.2, 11.3
ADOCommand	10.4.1
ADOConnection	10.4.1, 10.4.2, 10.4.3
ADODataSet	10.4.1
ADOQuery	10.4.1, 10.4.4
ADOStoredProc	10.4.1, 10.4.4
ADOTable	10.4.1, 10.4.2, 10.4.4
Animate	5.2.3
Application	1.5.3, 3.9.3, 4.1.9, 4.3.2.1, 4.5.1, 4.5.3, 4.5.4, 4.7.1.2, 6.1.2, 6.4.4.1, 6.5.3.5, 7.5.2, 12.10.1, 12.10.5.2, 15.6.2, 15.7.2.3
AppletApplication	2.4.2.2
AppletModule	2.4.2.2
ApplicationEvents	3.7.8, 3.9.3, 4.1.6.2, 4.1.9, 4.3.2.1, 4.7.1.2, 7.4
BatchMove	9.9
Bevel	3.7.2
BitBtn	3.5.1, 4.5.2, 5.1.2.3
Button	3.5.1, 4.3.1, 4.5.2
Ccalendar	3.3.3
CDirectoryOutline	3.8.3
Cgauge	3.4.3
Chartfx	3.4.7, 4.6.2
Chart	3.4.6, 4.6.2
CheckBox	3.5.4, 4.3.1.1, 7.3.4, 9.12
CheckListBox	3.2.5, 3.5.4
ColorDialog	3.8.5
ComboBox	3.2.5, 3.7.5, 4.5.6, 6.5.3.2, 9.5.5, 9.7, 9.12, 10.1.6.4, 10.1.7
ControlBar	3.7.6
CoolBar	3.7.6
CSpinEdit	3.3.2, 3.7.5, 6.3.4, 9.5.5, 13.4.1
Database	9.4, 9.8
DataModule	9.11.9
DataSource	9.4, 9.5.1, 10.1.6.1, 10.2.2, 10.4.1, 10.5.1
DateTimePicker	3.3.3
DBCheckBox	9.5.2, 9.7
DBComboBox	9.10.2
DBCtrlGrid	9.7, 9.10.1
DBEdit	9.4, 9.5.2, 9.7, 9.12, 11.3
DBGrid	9.4, 9.5.1, 9.5.5, 9.7, 9.8, 9.10.2, 10.1.6
DBImage	9.5.2, 9.7, 9.12, 11.3
DBListBox	9.10.2

DBLookupCombo	3.1
DBLookupComboBox	3.1, 9.10.2
DBLookupList	3.1
DBLookupListBox	3.1, 9.10.2
DBMemo	9.5.2, 9.7, 9.12
DBNavigator	9.5.1, 10.1.6.5
DBRadioGroup	9.7
DBRichEdit	9.7, 11.3
DBText	9.4, 9.7, 9.10.1
DDEClientConv	6.5.1, 6.5.2, 6.5.3.3, 6.5.3.4, 6.5.3.5
DDEClientItem	6.5.1, 6.5.3.2, 6.5.3.3, 6.5.3.4, 6.5.3.5
DDEServerConv	6.5.1, 6.5.2, 6.5.3.1, 6.5.3.5
DDEServerItem	6.5.1, 6.5.2, 6.5.3.1, 6.5.3.3, 6.5.3.4, 6.5.3.5
DecisionCube	11.1.1, 11.1.2
DecisionGraph	11.1.1, 11.1.3
DecisionGrid	11.1.1, 11.1.2
DecisionPivot	11.1.1, 11.1.3
DecisionQuery	11.1.1, 11.1.2
DecisionSource	11.1.1, 11.1.2
DirectoryListBox	3.8.3
DrawGrid	3.4.4
DriveComboBox	3.8.3
Edit	3.2.3, 3.7.6, 3.7.8, 3.8.3, 4.1.8, 4.2.5, 4.3.1.3, 4.3.2.2, 4.5.3, 5.1.2.2, 6.4.3, 7.2, 7.3, 7.5.3, 9.12
F1Book	3.3.4, 3.4.7
FileListBox	3.8.3
FilterComboBox	3.8.3
FindDialog	3.8.7
FontDialog	3.2.4, 3.8.4, 4.1.5, 4.7.2.1, 4.7.2.2, 6.4.3, 11.2
Form	1.5.3, 2.6.11, 4.2, 4.5, 5.1.1.2
Frame	3.7.8, 4.1.7, 7.4
Graph	3.4.7
GroupBox	3.5.3, 3.7.2, 3.7.8
Header	3.1, 3.7.3
HeaderControl	3.1, 3.7.3
HotKey	3.6.3
IBDatabase	10.5.1, 10.5.2, 10.5.3
IBDatabaseInfo	10.5.1
IBDataSet	10.5.1
IBEvents	10.5.1
IBQuery	10.5.1, 10.5.3
IBSQL	10.5.1
IBSQLMonitor	10.5.1
IBStoredProc	10.5.1, 10.5.3
IBTable	10.5.1, 10.5.3
IBTransaction	10.5.1, 10.5.2, 10.5.3
IBUpdateSQL	10.5.1
Image	4.4.3, 4.5.3, 4.6.4, 5.1.1–5.1.6, 5.2.2, 6.4.4.2, 6.4.4.3
ImageList	3.4.1, 3.4.2, 3.6.1, 3.7.4, 3.7.5, 3.7.6, 3.9.1, 3.9.2, 4.1.6.2
Label	3.2.2, 4.2.5
ListBox	1.5.6.2, 3.2.5, 3.6.3, 4.2.1, 4.2.2, 4.4.1, 4.7.1.2
ListView	3.4.2

MainMenu	3.4.2, 3.6.1, 3.6.3, 4.1.6, 4.5.4, 4.5.5, 4.7.1.2
MaskEdit	3.2.3
MediaPlayer	5.2.4
Memo	2.6.11, 3.2.4, 3.3.3, 3.8.4, 3.8.5, 3.8.7, 4.2, 4.4.1, 4.4.2, 6.1.2, 6.4.3, 6.4.4.3, 6.5.3.2, 6.5.3.3, 6.5.3.5, 10.1.7
MonthCalendar	3.3.3
NoteBook	3.1, 3.7.4
OLEContainer	6.4.2
OpenDialog	3.7.8, 3.8.2, 4.5.5, 5.2.1.2, 5.2.3, 5.2.4, 6.1.3, 6.4.2
OpenPictureDialog	3.8.2, 5.1.1.2, 5.1.1.4, 5.1.4, 5.1.7, 5.2.2
Outline	3.1, 3.4.1
PageControl	3.1, 3.7.4, 4.4.2, 9.12
PageScroller	3.7.5
PaintBox	5.1.7
Panel	3.2.2, 3.7.2, 3.7.4, 3.7.5, 4.1.9, 4.2.1, 4.2.2, 4.2.3, 4.2.4, 4.2.5, 4.4.2, 4.5.6
PopupMenu	3.6.2, 3.7.5, 4.1.6.1, 4.1.6.2
PrintDialog	3.8.6
Printer	4.6.4
PrinterSetupDialog	3.8.6
ProgressBar	3.4.3, 11.3
QRDBImage	11.2
QRDBRichText	11.2
QRDBText	11.2
QRImage	11.2
QRLabel	11.2
QRMemo	11.2
QRRichText	11.2
QRShape	11.2
QRSubDetail	11.2
QRSysData	11.2
Query	9.4, 9.8, 10.1.2.1, 10.1.2.2, 10.1.6, 10.1.7, 10.2.2, 10.2.4, 10.3.5.3, 10.4.1, 10.5.1, 11.2, 11.3
QuickRep	11.2
RadioButton	3.5.3, 3.7.2, 4.3.1.1
RadioGroup	3.5.3, 9.5.5
ReplaceDialog	3.8.7
RichEdit	3.2.4, 3.8.2, 3.8.4, 3.8.7, 4.1.6.2, 4.5.1, 4.5.5, 4.5.6, 4.6.2, 11.2, 13.9.1
SaveDialog	3.8.2, 4.5.5, 6.4.2, 6.4.3
SavePictureDialog	3.8.2, 5.1.6
Screen	4.1.5, 4.5.6, 5.1.2.4, 5.1.6
Scrollbar	3.5.5
ScrollBar	3.7.2
Session	9.4, 9.8, 9.11.9
Shape	3.4.5
SpeedButton	3.5.1, 3.5.2, 3.7.2, 5.1.2.3, 5.1.4, 6.5.3.2
Splitter	3.7.2, 4.2.3
StaticText	3.2.2, 4.2.1, 4.2.2
StatusBar	3.7.7, 3.9.3, 4.1.6.2, 4.1.9, 4.7.1.2, 11.3
StoredProc	9.4, 10.3.5.2, 10.3.5.3, 10.4.1, 10.5.1
StringGrid	3.2.6, 3.4.4

TabbedNoteBook	3.1, 3.7.4
TabControl	3.1, 3.7.4
Table	9.4, 9.5, 9.6, 9.7, 9.8, 9.9, 9.10, 9.11, 9.12, 10.1.2.1, 10.1.6.1, 10.1.7, 10.2.2, 10.2.4, 10.4.1, 10.5.1, 11.2
TabSet	3.1, 3.7.4
Timer	3.5.6, 4.5.3, 5.2.2
ToolBar	3.7.5, 3.7.6, 4.1.6.2, 11.3
TrackBar	3.5.5
TreeView	3.1, 3.4.1
UpdateSQL	10.1.6.5
UpDown	3.3.2
WordApplication	6.4.4, 11.3
WordDocument	6.4.4.2, 6.4.4.3, 11.3
WordFont	6.4.4.2, 6.4.4.3, 11.3
WordLetterContent	6.4.4.2
WordParagraphFormat	6.4.4.2, 6.4.4.3, 11.3

Литература

Более подробные сведения по многим вопросам, рассмотренным в данной книге, вы можете найти в книгах серии «Все о C++Builder»:

1. Архангельский А.Я. Интегрированная среда разработки C++Builder 5 — М: ЗАО «Издательство БИНОМ», 2000
2. Архангельский А.Я. Библиотека C++Builder 5: 70 компонентов ввода / вывода информации — М: ЗАО «Издательство БИНОМ», 2000
3. Архангельский А.Я. Библиотека C++Builder 5: 60 управляющих компонентов — М: ЗАО «Издательство БИНОМ», 2000
4. Архангельский А.Я. Разработка прикладных программ для Windows в C++Builder 5 — М: ЗАО «Издательство БИНОМ», 2000
5. Архангельский А.Я. Функции C++, C++Builder 5 и API Windows — М: ЗАО «Издательство БИНОМ», 2000
6. Архангельский А.Я. Язык C++ в C++Builder 5 (справочное пособие) — М: ЗАО «Издательство БИНОМ», 2000
7. Архангельский А.Я. Работа с локальными базами данных в C++Builder 5 — М: ЗАО «Издательство БИНОМ», 2000
8. Архангельский А.Я. Язык SQL в C++Builder 5 — М: ЗАО «Издательство БИНОМ», 2000





ПРОГРАММИРОВАНИЕ В C++Builder 5

Методика построения:

- текстовых редакторов
- графических редакторов
- мультимедиа и мультимедиа
- приложений с базами данных
- справочных систем
- шаблонов и компонентов
- интерфейсов к внешним программам
- отчетов

Справочные данные по:

- языку C++
- функциям C++
- функциям API Windows
- компонентам C++Builder 5
- классам C++Builder 5
- свойствам компонентов
- методам компонентов

Книга содержит методические и справочные материалы по новой версии системы визуального объектно-ориентированного программирования C++Builder 5. Рассмотрены такие новые возможности C++Builder 5, как интернационализация приложений, компоненты-серверы COM, технологии доступа к данным ADO и InterBase Express. Дается методика построения прикладных программ, реализующих текстовые и графические редакторы, мультимедиа и мультимедиа, работу с базами данных, построение справочных систем, отчетов, интерфейсов к внешним программам. Справочная часть книги содержит материалы по языку C++, функциям C++, C++Builder и API Windows, компонентам и классам C++Builder, их свойствам, методам и событиям.



Диск содержит полноценную пробную версию C++Builder 5 Trial Edition, справку .hlp по C++ и C++Builder 5 на русском языке и примеры, рассмотренные в книге.

Файл справочной системы на русском языке, содержащий около 2000 входов, описывает свыше 500 функций, около 200 свойств, методов и событий компонентов, типы данных, исключения и многое другое.

ISBN 5-7989-0191-2



9 785798 901913



ПРОГРАММИРОВАНИЕ В

C++ Builder 5